RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Dominique Kaiser

Der Readout-Merger des Präprozessors des ATLAS Level-1 Kalorimeter Triggers

Diplomarbeit

HD-KIP-05-02

KIRCHHOFF-INSTITUT FÜR PHYSIK

Fakultät für Physik und Astronomie

Ruprechts-Karls-Universität Heidelberg

Diplomarbeit
Im Studiengang Physik

Vorgelegt von
Dominique Kaiser
Aus Konstanz

April 2002

Der Readout-Merger des Präprozessors des ATLAS Level-1 Kalorimeter Triggers

Die Diplomarbeit wurde ausgeführt von Dominique Kaiser am

Kirchhoff-Institut für Physik der Universität Heidelberg

unter Betreuung von

Herrn Prof. Dr. Karlheinz Meier

Der RemFPGA des Präprozessors des ATLAS Level-1 Kalorimeter Triggersystems

Thema dieser Arbeit ist der Entwurf und die Entwicklung des Readout-Mergers für den Präprozessor des Level-1 Kalorimeter Trigger des ATLAS-Detektors. Dieser Baustein des Präprozessors wird mit Hilfe FPGA Technologie realisiert. Aufbauend auf den Resultaten des Tests eines Prototyps wurde ein vollständig neues Design entworfen.
Die Aufgabe des so genannten RemFPGA ist die Steuerung und Konfiguration aller anderen Komponenten des Präprozessormoduls wie auch die Kompression und Weiterleitung von Readout-Daten des Präprozessors. Bei dem FPGA handelt es sich um einen Virtex-E 1000 der Firma Xilinx. In der HDL-Sprache Verilog wurde ein komplettes Design für diesen FPGA-Baustein entwickelt und mit Hilfe von Software-Simulationswerkzeugen ausgiebig getestet. Der erstellte Programmcode ist modular aufgebaut, um Anpassungen an zukünftige Gegebenheiten zu erleichtern.
Für die effektive Übertragung der anfallenden Readout-Daten wurde ein in früheren Arbeiten entwickeltes Kompressionsverfahren wesentlich verfeinert und weiterentwickelt.

The RemFPGA of the pre-processor of the ATLAS Level-1 Calorimeter trigger system

Subject of this work is the planning and implementation of the Readout-Merger that is to be used by the pre-processor of the Level-1 Calorimeter Trigger of the ATLAS-Detector. This part of the pre-processor is to be carried out with the help of FPFA technology. Based on the results of research done in the past with a prototype a totally new design was developed.
The RemFPGA is to control and configure all other components of the pre-processor module as well as to compress and transfer Readout data produced by the pre-processor. The used FPGA is a Virtex-E 1000 from Xilinx. Using the HDL language Verilog a complete design was developed for this FPGA chip und subsequently it was tested thoroughly with the help of software simulation utilities. The produced Sourcecode is very modular to allow easy and effortless adjustments to changed conditions in the future.
To allow for effective transfer of the accumulating Readout data an existing compression schema was refined and improved significantly.

Inhaltsverzeichnis

Einleitung

Im Rahmen der Grundlagenforschung der Physik werden immer detailliertere Modelle unseres Universums entwickelt. Diese Modelle und Theorien postulieren physikalische Vorgänge und Gesetzmäßigkeiten, die in „normalen" Experimenten nicht zugänglich und überprüfbar sind, da sie eine sehr hohe Energie der beteiligten und untersuchten Komponenten erfordern. Aus diesem Grund erfindet und baut die physikalische Forschung immer neue und leistungsfähigere Teilchenbeschleuniger, um in immer höhere Energiebereiche vordringen zu können. Auf Grund der so durchgeführten Experimente werden neue Erkenntnisse gewonnen und neue Theorien entwickelt. Diese benötigen für ihre Verifizierung jedoch in vielen Fällen neue Experimente bei noch höheren Energien.
Eine der momentan zu überprüfenden Theorien betrifft die Existenz des so genannten Higgs-Teilchens.
Mit aus diesem Grund wird in Genf am Institut für Hochenergieforschung CERN ein neuer Beschleuniger gebaut. Der *Large Hadron Collider* – LHC wird Protonen mit einer Energie von bis zu 7 TeV zur Kollision bringen. Mit Hilfe von vier komplexen Detektoren, die teilweise die Größe von mehreren Reihenhäusern übertreffen, werden diese Kollisionen untersucht. Einer dieser Detektoren ist der so genannte ATLAS-Detektor.
Selbst bei diesen extrem hohen Energien treten die gewünschten Ereignisse nur sehr selten auf. Ein Higgs-Teilchen wird beispielsweise nur bei einer von 10^{11} Kollisionen erzeugt. Der Großteil der anderen Kollisionen erzeugt Resultate, die schon mit schwächeren Beschleunigern ausführlich untersucht werden konnten. Daher verfügen die Detektoren über ein Triggersystem, welches die interessanten Ereignisse identifizieren und zur weiteren Analyse extrahieren soll. Das Triggersystem des ATLAS-Detektors besteht aus drei Stufen. Jede Stufe verringert die zu verarbeitende und zu analysierende Datenmenge und ermöglicht somit den späteren Triggerstufen eine immer ausführlichere Analyse. ATLAS besteht aus mehreren verschiedenen Detektor-Untertypen, welche jeweils andere Messungen vornehmen. Daher bestehen die verschiedenen Triggerstufen auch aus mehreren verschiedenen Unterkomponenten.
Diese Arbeit beschäftigt sich mit einem Teil des so genannten Präprozessors des Level-1 Kalorimeter Triggers, also einem Teil der ersten Triggerstufe. Dieser Präprozessor arbeitet mit 7296 Messsignalen des Kalorimeter-Detektors. Er besteht aus einer Vielzahl von identischen Komponenten, genannt Präprozessormodule, welche jeweils 64 Messsignale verarbeiten.
Jedes dieser Module besitzt einen Baustein zur Steuerung aller Unterkomponenten dieses Moduls und zur Kommunikation mit der Außenwelt. Dieser so genannte Readout-Merger oder RemFPGA ist Gegenstand dieser Diplomarbeit.
Kapitel 1 erläutert die Motivation zum Bau des LHC und die grundlegende Funktionsweise des ATLAS-Detektors. Kapitel 2 erklärt die Funktion des ATLAS Triggersystems. Die Arbeitsweise des Präprozessorsystems wird in Kapitel 3 dargelegt. Kapitel 4 gibt einen Einblick in die für den RemFPGA verwendete Hardware. Kapitel 5 erläutert die Funktionsweise und das im Rahmen dieser Arbeit entwickelte interne Design des RemFPGA. Kapitel 6 beschreibt die Anwendung und Leistung des RemFPGA. Kapitel 7 widmet sich den Tests des entwickelten RemFPGA und Kapitel 8 fasst die Resultate dieser Arbeit zusammen und gibt einen Ausblick auf die weiteren Schritte in der Entwicklung des ATLAS Systems.

Kapitel 1

LHC und das ATLAS Experiment

Dieses Kapitel gibt einen Überblick über den physikalischen Kontext für den *Large Hadron Collider LHC* und das *Atlas*-Experiment (*A Toroidal LHC Apparatus)* entwickelt wird und das Experiment selbst.

1.1 Physikalischer Kontext

Die fundamentale Beschreibung unserer Welt und des Universums folgt letztendlich aus dem theoretischen Grundgerüst, welches die fundamentalen Teilchen und ihre Interaktionen beschreibt. Dieses Grundgerüst wird zurzeit durch das so genannte *Standardmodell* repräsentiert.

1.1.1 Das Standardmodell

Das Standardmodell ist das Resultat von ca. 50 Jahren theoretischer und experimenteller Forschung in der Teilchenphysik. Die immer genaueren und neuen experimentellen Befunde der letzten Jahrzehnte konnten letztlich immer mit dem Standardmodell erklärt werden oder das Modell konnte an die neuen Ergebnisse durch Verfeinerungen angepasst werden.
Das Standardmodell vereinheitlicht drei der vier elementaren Wechselwirkungen: Die starke, die schwache und die elektromagnetische. Die Gravitation ist nicht einbezogen, da sie zu schwach ist, um auf die bisher durchgeführten Experimente eine messbare Auswirkung zu haben.
Bis heute wurde noch kein einziges eindeutiges Resultat eines physikalischen Experiments gefunden, welches das Standardmodell widerlegen könnte. Dieser Abschnitt soll lediglich einen groben Überblick vermitteln und auf Schwächen des Standardmodells hinweisen.

Fundamentale Teilchen

Es gibt zwei grundsätzliche Klassen fundamentaler Teilchen. Es gibt *Fermionen* (Spin ½) und *Vektorbosonen* (Spin 1). Dabei vermitteln die Vektorbosonen die drei Grundkräfte, indem sie zwischen Teilchen ausgetauscht werden. Die Fermionen unterteilen sich weiter in *Quarks* und *Leptonen*. Diese Fermionen kann man anhand der aufsteigenden Masse der Teilchen in drei so genannte *Familien* oder *Generationen* einordnen.

Vektorbosonen				
Wechselwirkung	koppelt an	Typ	Masse	rel. Stärke
stark	Farbe	8 Gluonen	0	20
elektromagnetisch	el. Ladung	Photon	0	1
schwach	schwache Ladung	$W^{\pm}$, Z^0	~ 100GeV	10^{-7}
Gravitation	Masse	Gravitron	0	10^{-36}

	Quarks				Leptonen		
Fam.	Typ	Ladung	Farbe	Masse [MeV]	Typ	Ladung	Masse
1	down	- ?	r, b, g	5- 16 MeV	$?_e$	- 1	0,05- 1,5 eV
	up	+?	r, b, g	2- 8 MeV	e^-	0	511 eV
2	strange	- ?	r, b, g	˜ 100 MeV	$?_\mu$	- 1	0,05- 8.4 eV
	charm	+?	r, b, g	˜ 1,3 GeV	μ^-	0	105.7 MeV
3	bottom	- ?	r, b, g	˜ 4,3 GeV	$?_t$	- 1	0,05- 8.4 eV
	top	+?	r, b, g	˜ 175 GeV	t^-	0	1777.1 MeV

Zu all diesen Teilchen gibt es auch entsprechende Antiteilchen mit gleicher Masse, aber entgegengesetzter elektrischer Ladung, Farbe und schwacher Isospin.
Für die Gravitation fügt man eine vierte Wechselwirkung mit dem Gravitron als Vektorboson hinzu, welches an die Masse koppelt. Es gibt allerdings noch keine eindeutigen Beweise für die Existenz des Gravitrons.
Warum es gerade drei Familien gibt, ist unbekannt. Seit der Entdeckung der Neutrinomasse ist es auch denkbar, dass es noch weitere Familien gibt.
All diese Teilchen werden als fundamental und nicht weiter unterteilbar angesehen, sie besitzen keine innere Struktur und keine Ausdehnung.

Eigenschaften der fundamentalen Teilchen und ihre Symmetrien

Alle diese Teilchen werden im Standardmodell (und darauf aufbauenden Modellen) mit verschiedenen Eigenschaften versehen. Zu diesen Eigenschaften gehören die Masse, die elektrische Ladung, die schwache Ladung, der Spin, der schwache Isospin, die Farbe und Weitere mehr. Manche dieser Werte wie zum Beispiel die Masse haben (je nach Teilchen) einen bestimmten Wert aus einem kontinuierlichen Spektrum. Andere Eigenschaften wie zum Beispiel der Spin können lediglich diskrete Werte annehmen.
Diese diskreten Eigenschaften unterliegen speziellen mathematischen Symmetriebedingungen. Diese Bedingungen erlauben es, einen Zusammenhang zwischen verschiedenen fundamentalen Teilchen herzustellen und sie als verschiedene Ausprägungen ein und desselben „*Grundobjekts*“ anzusehen. Diese Ausprägungen werden auch Zustände genannt. Sie können mathematisch als Vektoren in einem multidimensionalen Raum ausgedrückt werden, welche durch Symmetrieoperationen ineinander übergeführt werden können.
Vereinfacht ausgedrückt erfordert zum Beispiel die Existenz eines Teilchens mit schwachem Isospin 1 (das W^+ Vektorboson) auch die Existenz von einem weiteren Zustand mit dem Wert -1 (das W^- Vektorboson) und einem dritten Zustand mit dem Wert 0 (das so genannte W^0, zusammen bilden sie ein so genanntes Triplett). Weiterhin legt es auch die Existenz eines vierten Zustands mit nochmals dem Wert 0 nahe (das so genannte B^0, es bildet ein so genanntes Singulett).
Diese Symmetrieregeln besagen jedoch lediglich, dass es zwei verschiedene Zustände mit schwachem Isospin 0 gibt, also zwei verschiedene unabhängige Vektoren. Nun kann man jedoch aus zwei unabhängigen Vektoren durch eine Drehung (eine Linearkombination) wieder zwei unabhängige, andere Vektoren ableiten. Welche *Drehung* nun zu zwei Teilchen führt, die in der Natur existieren, ist wiederum nicht das Ergebnisse einer mathematischen Operation, sondern muss als weitere Konstante im Standardmodell enthalten sein.
Im hier angeführten Beispiel bestimmt der so genannte *Weinberg-Winkel* die Mischung aus W^0 und B^0 und beschreibt so das dritte Vektorboson der schwachen Kraft Z^0 und das Vektorboson der elektromagnetischen Kraft, das Photon (γ).

$$\begin{aligned} \left|W^+\right\rangle &= \left|W^+\right\rangle \\ \left|W^-\right\rangle &= \left|W^-\right\rangle \\ \left|Z^0\right\rangle &= \cos q_W \left|B^0\right\rangle + \sin q_W \left|W^0\right\rangle \\ \left|g\right\rangle &= -\sin q_W \left|B^0\right\rangle + \cos q_W \left|W^0\right\rangle \end{aligned}$$

Gleichzeitig wird auf diese Weise auch ein Zusammenhang zwischen der Einheit der elektrischen und der schwachen Ladung hergestellt und auch zwischen der relativen Stärke der beiden Kräfte. Somit kann die elektromagnetische Kraft und die schwache Kraft in der so genannten *elektroschwachen Vereinheitlichung* zusammengefasst werden.

Das postulierte Higgs-Boson

Leider hat die oben beschriebene *elektroschwachen Vereinheitlichung* einen gravierenden Schönheitsfehler:
Eine Vermischung von verschiedenen Zuständen ist nach den gegebenen Gesetzmäßigkeiten nur möglich, wenn die beteiligten Zustände vergleichbare Massen besitzen. Leider besitzt das W^0 jedoch eine nicht unerhebliche Masse von ca. 100GeV, während das Photon masselos ist.
Der Widerspruch kann durch die Postulierung von vier verschiedenen *Higgs-Feldern* gelöst werden. Jedem der vier Vektorbosonen wird ein solches Feld zugeordnet. Die Kraft des Feldes wird durch jeweils ein weiteres Boson übermittelt. Es gibt somit vier Higgs-Bosonen. Die postulierte Theorie besagt nun, dass unterhalb einer bestimmten Temperatur/Energie (welche jeweils für die vier Felder unterschiedlich sein kann) jeweils ein Higgs-Boson von einem Vektorboson absorbiert wird.
In der Higgs-Theorie sind alle vier Vektorbosonen masselos während alle vier Higgs-Bosonen eine Masse besitzen. Es wird nun weiter postuliert, dass bei der aktuellen „Temperatur“ des Universums die drei Higgs-Bosonen von W^+, W^- und Z^0 absorbiert werden und diesen somit eine Masse verleihen. Das Photon hat sein Higgs-Boson jedoch nicht absorbiert und ist daher masselos. Es muss also ein freies Higgs-Boson geben!
Die Masse dieses Higgs-Bosons lässt sich grob abschätzen und liegt in einem Energiebereich, der mit bisherigen Teilchenbeschleunigern nicht erreichbar ist.
Der Beweis der Existenz oder Nichtexistenz des Higgs-Teilchens ist von fundamentaler Bedeutung für die Konsistenz des Standardmodells. Bei einer Nichtexistenz wären völlig neue theoretische Konzepte erforderlich. Eine der Hauptmotivationen für den Bau des *Large Hadron Collider LHC* ist die Produktion und der Nachweis von Higgs-Bosonen.

1.2 Der Large Hadron Collider LHC

Der Large Hadron Collider wird gebaut um bei Proton—Proton Kollisionen Schwerpunktsenergien von bis zu 14 TeV bei einer Luminosität[1] von 10^{34} $cm^{-2}s^{-1}$ erzeugen zu können. Die Energiebereiche von bisherigen Beschleunigern liegen um Größenordnungen tiefer.
Der LHC besteht aus zwei getrennten Synchotronringen mit einem Durchmesser von 8.6 Kilometern. Er wird am CERN[2] bei Genf in den schon bestehenden Tunnel des beendeten LEP[3] Experiments installiert und wird voraussichtlich 2006 in Betrieb gehen. In den beiden Beschleunigungsringen können entweder Pakete von Protonen oder von Schwerionen beschleunigt und an vier verschiedenen Stellen zur Kollision gebracht werden. Wird der Beschleuniger mit Protonen betrieben, treffen die Pakete[4] alle 25 ns aufeinander, bei Schwerionen alle 100 ns. Beim Betrieb mit Protonen finden durchschnittlich pro Paketdurchdringung 25 Kollisionen statt. Die vier Kollisionspunkte sind die Standorte von unterschiedlichen Experimenten.
ATLAS[5] und **CMS**[6] stellen Mehrzweck-Experimente dar, die ungefähr die gleichen Zielsetzungen haben. Die jeweiligen Detektoren sind von der Technik her jedoch deutlich unterschiedlich konzipiert.

---

[1] Die Luminosität beschreibt die Anzahl an Teilchenkollisionen pro Fläche und Zeiteinheit
[2] CERN – Centre Européenne pour la Recherche Nucléaire
[3] LEP – Large Electron Proton Collider
[4] das Durchdringen der beiden Teilchenwolken nennt man „bunch-crossing“
[5] ATLAS – A Torodial LHC Apparatus
[6] CMS – Compact Muon Solenoid

ALICE[7] ist ein Schwerionen-Experiment und untersucht Kern-Kern-Wechselwirkungen bei sehr hohen Energien. Die Kollisionen finden hier unter Bedingungen statt wie sie mit dem Universum direkt nach dem Urknall vergleichbar sind.
LHC-B[8] untersucht CP-Verletzung beim Zerfall von B-Mesonen. Der Detektor untersucht ausschließlich die Vorwärtsstreuung bei sehr kleinen Winkeln, da in diesem Bereich B- und Anti-B-Mesonen besonders zahlreich auftreten.
Eine der Hauptaufgaben von LHC besteht in der Produktion und Detektion von Higgs-Bosonen. LHC wird der erste Detektor sein, der Energien erreicht, die zur Produktion von Higgs-Bosonen in großen Mengen nötig sind. Der Vorgänger von LHC das LEP-Experiment hat erste Indizien von Higgs-Bosonen geliefert, welche darauf hindeuten, dass die Masse des Higgs-Bosonen sich am unteren Ende des vorhergesagten Bereichs befindet. Daher sollte LHC Higgs-Bosonen in sehr großen Mengen erzeugen können.
Weiterhin wird nach bisher unbeobachteten supersymmetrischen Teilchen gesucht, welche einen Beleg für die Erweiterung des Standardmodells durch Supersymmetrie liefern würden.
Außerdem sollen mit Hilfe von LHC-B Präzisionsmessungen von CP-Verletzungen durchgeführt werden, um so einige Naturkonstanten des Standardmodells genauer zu bestimmen (Cabbibo-Kobayashi-Maskawa-Matrix). Darüber hinaus ist bei diesen hohen Energien eventuell auch mit der Produktion von bisher völlig unbekannten Teilchen zu rechnen, die das Standardmodell in seiner jetzigen Form überholen werden.
Abbildung 1 zeigt eine einfache historische Übersicht verschiedener Teilchenbeschleuniger und ihren maximalen Energien. Der LHC wird der erste Beschleuniger sein, welchem der gesamte Energiebereich der postulierten Higgs-Masse zugänglich sein wird. Abbildung 2 zeigt die verschiedenen Wirkungsquerschnitte am LHC und macht deutlich, dass nur etwa eine von 10^{11} Kollisionen ein Higgs-Teilchen erzeugen wird (vergleiche totalen Querschnitt σ_{tot} mit Higgs-Querschnitt σ_{Higgs}). Somit muss viel Sorgfalt auf die Erkennung einiger weniger besonderer Ereignisse innerhalb von vielen eher uninteressanten Ereignissen gelegt werden. Dies ist die Aufgabe eines so genannten Triggersystems (siehe Kapitel 2).

[7] ALICE – A Large Ion Collider Experiment
[8] LHC Bottom Quarks Experiment

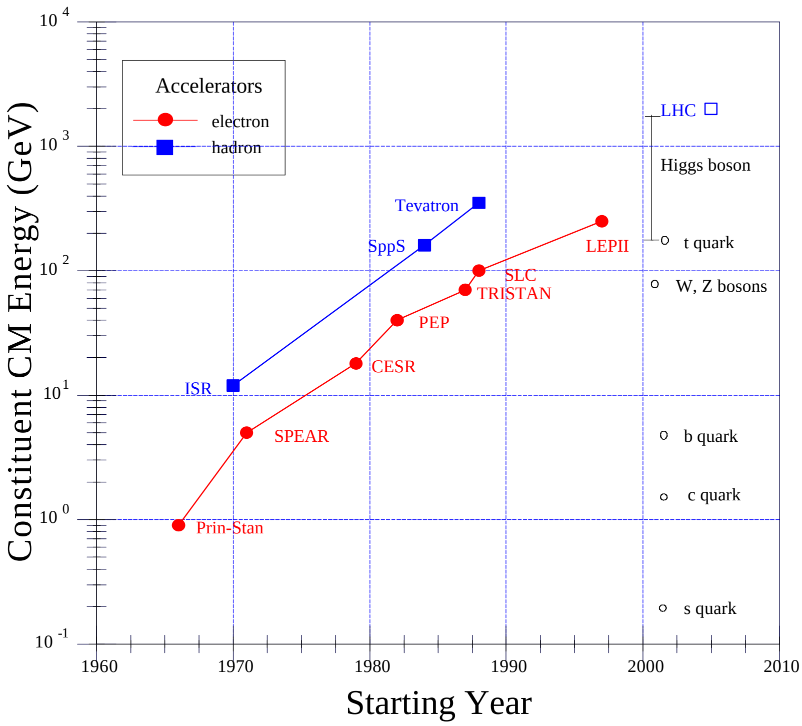


Abbildung 1 - Geschichte der Teilchenbeschleuniger

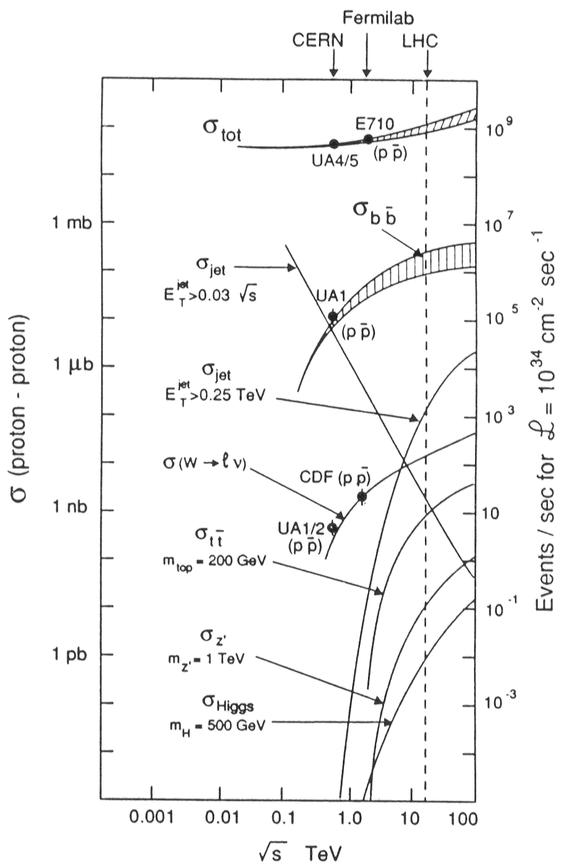


Abbildung 2 - LHC Proton-Proton Wirkungsquerschnitt

1.3 Das ATLAS-Experiment

Als einer der beiden Mehrzweck-Detektoren wurde ATLAS entworfen um durch Kollisionen erzeugte Teilchen in dem gesamten möglichen Energiebereich messen zu können. Dabei wurde sehr viel Wert auf eine möglichst hohe und genaue Auflösung der gemessenen Orts- und Energiewerte gelegt. Da LHC mit sehr hohen Luminositäten und Energien arbeitet, mussten Detektorkomponenten entwickelt werden, die die hohe entstehende radioaktive Strahlung über Jahre hinweg aushalten können. Vor allem integrierte elektronische Schaltungen sind sehr strahlungsempfindlich. Dies hatte maßgeblichen Einfluss auf das Grunddesign des Detektors.
Der Detektor an sich besteht aus mehreren eigenständigen Komponenten, welche im Folgenden kurz erläutert werden. Der Detektor ist in Abbildung 3 schematisch dargestellt.

1.3.1 Der Innere Detektor

Der Innere Detektor hat die Aufgabe, den Ursprungsort einer Kollision und den Weg der dabei produzierten Teilchen mit extrem hoher Genauigkeit zu messen. Da sich der Detektor darüber hinaus auch in einem Magnetfeld parallel zur Strahlachse befindet, können mit Hilfe von Energiemessungen der weiter außen liegenden Kalorimeter auch die Impulse von geladenen Teilchen bestimmt werden.
Der innerste Teil dieses Detektors besteht aus Schichten von Halbleiter-Bausteinen zur punktgenauen Detektion. Etwas weiter außen befinden sich Halbleiterschichten zur Streifen-Detektion, und zum Schluss folgen Gaszähler (Drahtkammern gefüllt mit Xenongas). Halbleiter-Detektoren werden nicht überall eingesetzt, da sie einerseits einen relativ hohen Wirkungsquerschnitt besitzen und somit bei verstärkter Verwendung Energie absorbieren und die Kalorimetermessung verfälschen. Andererseits sind diese Detektoren sehr teuer, und es ist nicht vollständig gesichert wie resistent diese Bauteile gegenüber jahrelanger harter Strahlung sind.
Bei einer typischen Kollision durchläuft jedes Streuteilchen drei Punktschichten, acht Streifenschichten und 36 Drahtkammern, was eine sehr hohe Genauigkeit der Orts- und Impulsmessung ermöglicht.
Insgesamt verfügt der Innere Detektor über rund 146 Millionen Kanäle.

1.3.2 Das Kalorimeter

Die Aufgabe des Kalorimeters ist die Messung der Energie aller in einer Kollision erzeugten Teilchen. Lediglich Myonen und Neutrinos können nicht gemessen werden, da sie das Kalorimeter durchdringen ohne Spuren zu hinterlassen. Da das Kalorimeter aus mehreren Schichten besteht, können auch die Richtungen der gemessenen Teilchen bestimmt werden.
Diese Schichten bestehen abwechselnd aus einem Absorbermaterial und detektierendem Material. Die zu messenden Teilchen verlieren in den Absorberschichten Energie, was zur Paarproduktion von weiteren Teilchen führt. Diese sekundären Teilchen produzieren auf ihrem Weg durch das Kalorimeter wiederum weiter Teilchen. Auf diese Weise entsteht ein so genannter Teilchenschauer. Je nach Typ der primären Teilchen entstehen unterschiedliche Arten von Schauern. In den detektierenden Schichten erzeugen all diese Teilchen Signale proportional zu ihrer Energie. Die Detektionsschichten sind in eine Vielzahl von einzelnen Zellen unterteilt, was die räumliche Auflösung innerhalb des Kalorimeters ermöglicht. Jede dieser Zellen liefert ein Signal, dessen Stärke proportional zur Energie des gemessenen Teilchens ist.
Eine Teilaufgabe des Kalorimeters ist also die möglichst starke Absorption und Paarproduktion von Teilchen. Dies bedeutet, dass das gesamte Kalorimeter einer sehr starken Strahlung ausgesetzt ist, wodurch sehr hohe Anforderungen an die verwendeten Materialen und elektronischen Schaltungen gestellt werden.
Das Kalorimeter des ATLAS Experiments besteht aus drei verschiedenen Einheiten:

Das elektromagnetische Kalorimeter

Das EM Kalorimeter dient der Messung von Elektronen und Photonen. Als Absorptionsmaterial wird Blei verwendet, welches von Hadronen relativ ungehindert durchdrungen wird, Elektronen und Photonen dagegen erzeugen starke Bremsstrahlung. Als Detektionsmaterial wird flüssiges Argon verwendet, welches gute Messeigenschaften und hohe Unempfindlichkeit gegenüber der starken Strahlung in sich vereint.
Das elektromagnetische Kalorimeter besteht aus über 200.000 einzelnen Zellen, was eine extrem hohe räumliche Auflösung erlaubt.

Das hadronische Kalorimeter

Als Absorptionsmaterial für Hadronen wird Eisen (Stahl) verwendet, zur Detektion werden Plastikszintillatoren eingesetzt.
Die Absorption von hadronischen Teilchen ist wesentlich schwächer als im Vergleich die von Elektronen oder Photonen. Daher ist das hadronische Kalorimeter wesentlich dicker als das EM Kalorimeter.
Die Szintillatoren bestehen aus etwa 35.000 einzelnen Zellen.

Das Vorwärts-Kalorimeter

Für Teilchen mit Bahnen nahe an der Strahlachse gibt es das so genannte Vorwärts-Kalorimeter. Da in diesem Streubereich eine besonders starke Strahlung herrscht, wird als Detektionsmaterial wieder flüssiges Argon eingesetzt. Etwa 11.000 Zellen dienen der räumlichen Auflösung.

1.3.3 Das Myon-Spektrometer

Da Myonen nur elektroschwach wechselwirken, durchdringen sie das gesamte Kalorimeter fast ungehindert.
Das gesamte Kalorimeter ist von supraleitenden Magneten umgeben, welche ein zylindersymmetrisches (toroidales) Magnetfeld erzeugen. Die von dem Feld abgelenkten Myonbahnen werden durch Driftkammern detektiert.
Da der größte Teil der restlichen Teilchen im Kalorimeter absorbiert wird, ist das Myon-Spektrometer nur einer wesentlich schwächeren Strahlung ausgesetzt. Dies macht die Verwendung von wesentlich komplexeren elektronischen Schaltungen möglich.
Die Driftkammern des Myon-Spektrometers produziert Messdaten aus ungefähr 440.000 einzelnen Kanälen.

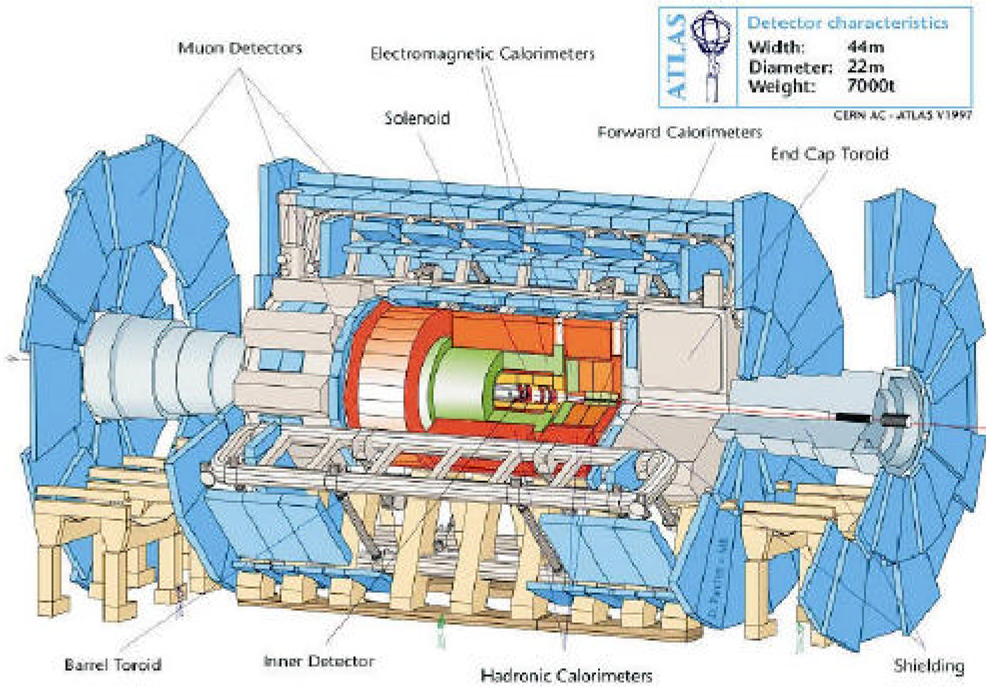


Abbildung 3 - Schema des ATLAS Detektors

Kapitel 2

Das ATLAS Triggersystem

Alle Kanäle aller Detektorkomponenten des ATLAS Experiments produzieren alle 25ns[9] Daten, die komplett zur weiteren Verarbeitung und Speicherung ausgelesen werden müssen. Dies ergibt ungefähr 6×10^{15} Einzelsignale pro Sekunde. Eine solche extreme Datenmenge kann weder mit heutigen noch mit für die nähere Zukunft absehbaren Technologien auf Datenträgern abgespeichert werden. Da wegen der starken Strahlung direkt am Detektor nur einfache und robuste elektronische Komponenten verwendet werden können, ist es sogar unmöglich die gesamte Datenmenge mit der nötigen Geschwindigkeit vom Detektor zu einer weiteren Verarbeitungseinheit zu übertragen.
Nicht jede Proton-Proton Kollision erzeugt jedoch Ergebnisse von physikalischem Interesse.
Aus diesem Grund verwendet man einen so genannten Trigger, um aus der großen Menge von Ereignissen nur diejenigen von Interesse auszufiltern. Nur im Falle einer positiven Triggerentscheidung werden sämtliche Kanäle des ATLAS Detektors ausgelesen und zur späteren Analyse gespeichert.
Eine solche Triggerentscheidung kann nicht innerhalb von 25ns getroffen werden und daher müssen die Daten aller Kanäle in Pipeline-Memories auf dem Detektor zwischengespeichert werden. Aus Kostengründen und aus Gründen der technischen Machbarkeit dieser Pipeline-Memories muss die Latenzzeit des Triggersystems möglichst kurz gehalten werden.
Das Triggersystem des ATLAS Experiments besteht aus drei nacheinander geschalteten Entscheidungsschritten: Der Level-1 Trigger, der Level-2 Trigger und der Eventfilter. Jeder der drei Einzeltrigger verringert die Datenmenge für die folgenden Triggersysteme.
Der Level-1 Trigger arbeitet auf der vollen LHC Rate von 25ns. Diesem Triggersystem steht eine fest definierte Zeit (Latenz) zur Entscheidung zur Verfügung. Bei einem positiven Resultat werden die Detektordaten bis zur Entscheidung des Level-2 Triggers aus den Pipeline-Memories in einen Readout Speicher übertragen. Auf diese Weise wird die Größe der kostspieligen Pipeline-Memories minimiert. Der Eventfilter schließlich rekonstruiert und analysiert das komplette ATLAS Event und entscheidet, ob dieses von physikalischem Interesse ist und abgespeichert werden soll.
Der Level-1 Trigger hat die Aufgabe, die Datenrate von 40MHz auf maximal 100kHz zu reduzieren. Der Level-2 Trigger reduziert die Rate nochmals auf maximal 1kHz. Der Eventfilter schließlich akzeptiert Ereignisse mit einer Höchstrate von 100Hz. Die verbleibende Datenmenge ist noch immer sehr groß, kann jedoch mit der verfügbaren Hardware bewältigt werden.

2.1 Der Level-1 Trigger

Die Aufgabe des Level-1 Triggers ist es, mithilfe einer möglichst schnellen Entscheidungslogik die Datenrate um den Faktor 400 zu reduzieren. Der Entscheidungslogik steht dabei ein Zeitraum von 2µs zur Verfügung (inklusive Verzögerung durch Länge von Leitungen) was 80 bunch-crossings (Proton-Proton-Durchdringungen) entspricht.
Um diesen Zeitrahmen einhalten zu können arbeitet der Level-1 Trigger mit Messdaten von reduzierter Genauigkeit des Myon-Spektrometers und des Kalorimeters. Die Entscheidungslogik wird von spezieller Hardware implementiert, wofür hauptsächlich ASICs[10] und FPGAs[11] eingesetzt werden.

[9] 25 ns beim Betrieb mit Protonen, 100 ns beim Betrieb mit Schwerionen
[10] ASIC – Application Specific Integrated Circuit, eigens entwickelte Hardware Chips

Der Level-1 Trigger ist in mehrere unabhängige Untereinheiten unterteilt. Die Entscheidungen der einzelnen Subsysteme werden vom Central Trigger Prozessor (CTP) zusammengeführt und verarbeitet.

2.1.1 Der Trigger des Kalorimeters

Um die Eingangsdatenmenge zu reduzieren werden die Daten der etwa 250.000 Zellen des hadronischen und elektromagnetischen Kalorimeters zu 7296 gröberen Zellen summiert. Dies geschieht mit analogen Schaltungen direkt auf dem Detektor. Die summierten Signale werden analog elektrisch über rund 60m Kabel zur Level-1 Trigger Elektronik übertragen. Der Kalorimeter Trigger erfüllt folgende Aufgaben:

- Vorverarbeitung der Eingangssignale (Pre-processing, siehe Kapitel 3)
- Triggerentscheidung für Elektronen und Photonen
- Triggerentscheidung für einzelne Hadronen und Tau-Teilchen
- Triggerentscheidung für Teilchen-Jets
- Berechnung der gesamten Energie im Kalorimeter um eine Entscheidung aufgrund der fehlenden Transversalenergie zu treffen

Die Vorverarbeitung erfolgt im so genannten Präprozessor. Hier werden die Signale digitalisiert und kalibriert. Außerdem analysiert der Präprozessor die Eingangssignale um das bunch-crossing zu identifizieren, welches das Signal verursacht hat. Diese Daten werden weitergeleitet an den Cluster Prozessor und den Jet/Energy-Sum Prozessor. Eine genauere Erläuterung des Präprozessors folgt in Kapitel 3.
Der Cluster Prozessor (CP) erkennt einzelne Elektronen, Photonen und Hadronen oberhalb einer definierbaren Energie. Hierzu wird eine 4x4 Nachbarschaftsmatrix mit der Matrix der 7296 Eingangssignale multipliziert und auf diese Weise nach isolierten Energiemessungen gesucht.
Der Jet/Energy-Sum (JEP) Prozessor erkennt Jets und berechnet die fehlende transversale Energie. Hierzu wird eine größere Matrix mit der Eingangsmatrix multipliziert. Die Matrix ist so gewählt, dass nur Zellen erkannt werden in deren direkter Nachbarschaft sich weitere Zellen mit positiver Energiemessung befinden.
Die Anzahl der erfüllten Bedingungen (z.B. überschrittene Mindestenergien) wird an den Central Trigger Prozessor übermittelt, welcher die Level-1 Trigger Entscheidung trifft. Bei einer positiven Level-1 Entscheidung (Level-1 Accept Signal) werden Regions-of-Interest (RoI) Informationen an den Level-2 Trigger übertragen. Dies sind die Regionen, die zum Level-1 Accept geführt haben und vom Level-2 Trigger weiter analysiert werden.
Um die korrekte Funktionsweise und Kalibrierung des Kalorimeter Level-1 Triggers gewährleisten zu können, liefern alle drei Subsysteme zusätzlich zu den direkten Entscheidungsdaten noch so genannte Readout Daten. Der Präprozessor ermöglicht die Auslese von Rohdaten direkt nach der Digitalisierung und Daten nach der Vorverarbeitung. CP und JEP liefern ihre jeweiligen Eingangsdaten und die Ergebnisse der Analyse.
Der Datenpfad des gesamten Level-1 Kalorimeter ´Triggers ist in Abbildung 4 dargestellt.

[11] FPGA – Field Programmable Gate Array, siehe Kapitel 4

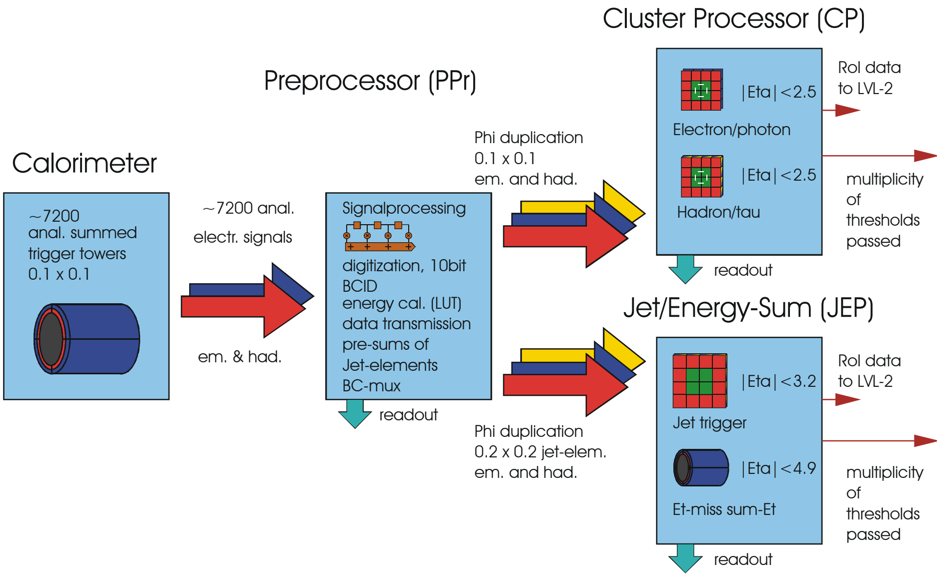


Abbildung 4 - Schema der Level-1 Kalorimeter Triggers

2.1.2 Der Myontrigger

Auch der Myontrigger arbeitet nicht mit der vollen Granularität des Detektors. Die Eingangsdaten des Myontriggers stammen jedoch im Gegensatz zum Kalorimetertrigger nicht vom normalen Myondetektor (Driftkammern), sondern von spezieller Detektorhardware, die nur für die Triggerentscheidung verwendet wird (Resistive-Plate Chambers und Thin-Gap Chambers).
Der Myontrigger verfügt über sechs verschiedene programmierbare Grenzwerte für Myonimpulse. Wie auch beim Kalorimetertrigger wird die Anzahl der erfüllten Bedingungen an den Central Trigger Prozessor übertragen und im Falle eines Level-1 Accept Signals die RoI Informationen an den Level-2 Trigger.

2.1.3 Der Central Trigger Prozessor

Der Central Trigger Prozessor (CTP) erfasst sämtliche Entscheidungsdaten des Kalorimetertriggers, des Myontriggers und einer optionalen externen Datenquelle. Diese Daten werden mit 96 verschiedenen programmierbaren Bedingungen verglichen. Erfüllen die Daten mindestens einen der 96 Bedingungen wird ein Level-1 Accept Signal generiert.
Eine einzelne Bedingung kann zum Beispiel erfordern, dass zwei bestimme Myon-Grenzwerte nicht erfüllt sein dürfen, aber bestimmte Elektron- oder Hadron-Grenzwerte überschritten sein müssen.
Ein positives Level-1 Accept wird über das globale ATLAS Timing, Trigger and Control System (TTC) im gesamten System verteilt. Es enthält Informationen des ausgelösten Trigger Typs, der identifizierten bunch-crossing Nummer und der fortlaufenden Nummer des Level-1 Events. Auch die untergeordneten Level-1 Trigger Subsysteme erhalten diese Informationen und nutzen diese unter anderem um Readout Daten zur optionalen Auslese verfügbar zu machen, so dass die Trigger-Entscheidung nachträglich auf Ihre Richtigkeit überprüft werden kann.

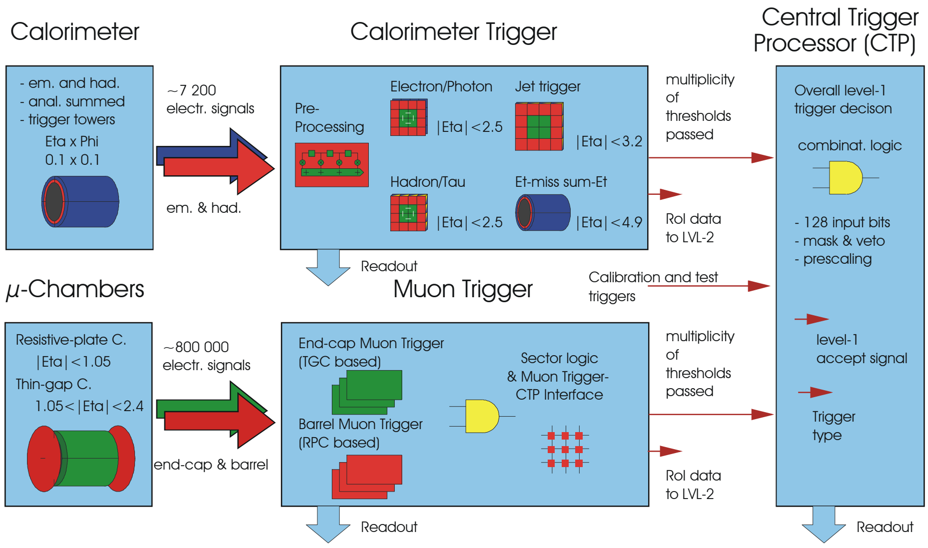


Abbildung 5 - Schema der ATLAS Level-1 Triggers

2.2 Der Level-2 Trigger

Die Aufgabe des Level-2 Triggers ist es, die Datenrate um den Faktor 100 auf 1kHz zu reduzieren. Hierzu erhält der Level-2 Trigger Region-of-Interest (RoI) Informationen von der Level-1 Trigger Logik. Mit der Hilfe dieser Informationen wählt der Level-2 Trigger die entsprechenden Zellen des ATLAS Detektors und liest diese aus dem Readout Speicher mit der vollen Genauigkeit aus.
Die Logik verwendet diese Daten um, so weit als möglich Bahn, Impuls, Masse und Typ von Teilchen zu bestimmen. Dies ist nicht immer vollständig möglich, da auch hier die zur Verfügung stehende Zeit stark begrenzt ist. Die auf diese Weise identifizierten Teilchen werden dann ähnlich wie im Level-1 Central Trigger Prozessor mit einer Anzahl von Sätzen von Bedingungen verglichen und bei Erfüllung mindestens einer Bedingung erfolgt das Level-2 Accept Signal. Eine Bedingung könnte zum Beispiel ein auftretendes Higgs-Teilchen sein.
Auch für den Level-2 Trigger wird dedizierte Hardware in Form von FPGAs eingesetzt.

2.3 Der Eventfilter

Die Aufgabe des Eventfilters ist eine weitere Reduktion der Datenrate um den Faktor 10. Der Eventfilter wird aus einer Farm von kommerziell erhältlichen Prozessoren bestehen. Er wird über ausreichend Rechenkraft verfügen, um anhand der kompletten ATLAS Detektordaten das erfolgte Event vollständig zu rekonstruieren. Das analysierte Event wird dann wiederum mit mehreren Sätzen von Bedingungen verglichen und bei Übereinstimmung zur späteren offline Analyse abgespeichert.
Der Eventfilter befindet sich momentan noch in der Planungsphase, da die Prozessorfarm mit der leistungsfähigeren Hardware der Jahre 2005 und 2006 implementiert werden soll. Heutige Hardware kann die erforderliche Rechenkraft mit den verfügbaren Geldmitteln nur schwer bis gar nicht leisten. Bei weiterer Gültigkeit von Moores Gesetz wird die nötige Leistung bis zum Jahr 2005 allerdings problemlos erhältlich sein.

In Abbildung 6 ist der gesamte Datenpfad des ATLAS Experiments dargestellt.

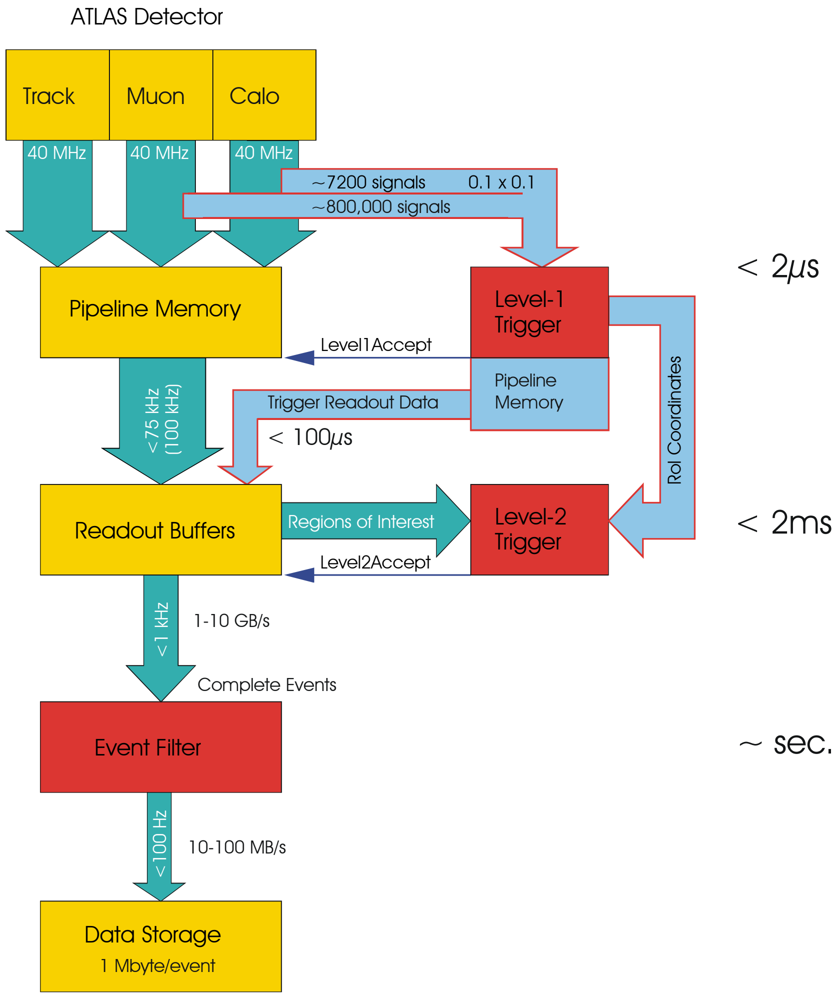


Abbildung 6 - Datenpfad der ATLAS Triggersystems

Kapitel 3

Der Level-1 Kalorimetertrigger Präprozessor

Der Präprozessor erhält als Eingang alle 25 Nanosekunden 7296 analoge Signale vom ATLAS Kalorimeter. Diese 7296 Signale werden direkt auf dem Detektor durch analoge Summation aus den Signalen der über 200.000 Zellen des elektromagnetischen und hadronischen Kalorimeters generiert. Nur durch diese stark reduzierte Granularität ist es dem Level-1 Trigger möglich mit einer ausreichenden Geschwindigkeit Entscheidungen zu treffen.
Der Präprozessor ist die erste signalverarbeitende Instanz und hat als solche die Aufgabe die eingehenden analogen Signale zu digitalisieren und zu kalibrieren. Darüber hinaus liefert der Präprozessor über Readout-Daten die Möglichkeit die Integrität des Level-1 Trigger Systems zu überprüfen. Insbesondere können die unverarbeiteten Rohdaten direkt nach der Digitalisierung ausgelesen werden. Hiermit kann man die korrekte Funktion der analogen Summation und die korrekte Kalibrierung der Daten überprüfen. Auch die Zuverlässigkeit der Algorithmen zur Identifikation des Bunch-Crossings kann getestet werden.
Im Folgenden hat der Präprozessor folgende Detailaufgaben:

3.1 Aufgaben des Präprozessors

Wie erwähnt ist die gesamte, alle 25 Nanosekunden vom ATLAS Detektor generierte Datenmenge zu groß um direkt bewältigt werden zu können. Daher kommt dem gesamten Level-1 Trigger System eine besonders wichtige Bedeutung zu. Es ist die erste Instanz in der gesamten Triggerkette und *muss* ordnungsgemäß arbeiten. Sollte der Level-2 Trigger oder der Eventfilter ausfallen, kann man als Notbehelf den Level-1 Trigger mit hohen Energiegrenzwerten betreiben. Die auf diese Weise stark reduzierte Level-1 Accept Rate kann dann als behelfsmäßiges endgültiges Triggersignal verwendet werden. Umgekehrt jedoch sind ohne Level-1 Trigger der Level-2 Trigger und der Eventfilter völlig nutzlos.
Grundsätzlich existieren die folgenden Teilaufgaben des Präprozessors:

- **Empfang des differentialen analogen Signals**
  Das Signal muss in ein unipolares Signal umgewandelt werden. Ein programmierbarer 10-Bit DAC erlaubt die Justierung des Nullniveaus.
- **Digitalisierung des analogen Signals**
  Ein kommerzieller 10-Bit Flash ADC wird verwendet um das Signal mit der globalen LHC-Frequenz von etwa 40MHz zu
  digitalisieren.
- **Phasenjustierung des Eingangssignals**
  Mit einem ASIC (Phos4) der CERN Microelectronics Group kann die Clock des FADCs gegenüber der LHC-Clock in Schritten von einer Nanosekunde in der Phase verschoben werden.

- **Identifikation des Bunch-Crossings**[12]
  Das eingehende Signal muss einem bestimmten Bunch-Crossing zugeordnet werden, damit bei einem Level-1 Accept die Detektordaten des korrekten Events ausgelesen werden.
  Dies erfolgt mit Hilfe zweier verschiedener Algorithmen, einer für gesättigte und einer für ungesättigte Signale. Diese Analyse wird von einem dedizierten ASIC durchgeführt, dem Pre-Processor ASIC (PPrASIC).
  Zusätzlich kann ein so genanntes „External BCID“ Signal zur Identifikation verwendet werden.

- **Kalibrierung des digitalisierten Signals**
  Über eine frei konfigurierbare Lookup-Tabelle wird das 10-Bit Rohsignal (RAW) auf einen 8-Bit Wert abgebildet (nach der BC Identifikation)

- **Berechnung von Jetelementen**
  Der Jet/Energy-Sum Trigger arbeitet mit nochmals gröberen Eingangsdaten. Hierzu werden jeweils 4 der 7296 vorverarbeiteten Signals aufsummiert.

- **Serielle differentielle Datenübertragung**
  Die vorverarbeiteten Daten werden über eine serielle LVDS[13]-Verbindung an den Cluster Prozessor und den Jet/Energy-Sum Prozessor zur weiteren Verarbeitung übertragen.

- **Readout von Trigger Daten**
  Um die korrekte Funktion des Präprozessors zu überprüfen können die unverarbeiteten Rohdaten (RAW) ausgelesen werden.

- **Readout von Histogramm- und Ratemeter-Informationen**
  Der PPrASIC verfügt zusätzlich über die Möglichkeit ein Histogramm und Ratenmessungen zu erstellen. Auch diese Informationen müssen bei Bedarf ausgelesen werden.

3.2 Design des Präprozessorsystems

Dem gesamten Level-1 Trigger stehen 80 Bunch-Crossings (2µs) an Zeit zur Verfügung um eine Einscheidung zu treffen, dies beinhaltet Signalverzögerungen durch Kabelverbindungen. Dem Präprozessor des Triggers stehen maximal 18 Bunch-Crossings zur Verfügung. Diese sehr kurze Zeitspanne macht den Einsatz von hochintegrierten Schaltungen wie dem PPrASIC (Pre-Processor ASIC) nötig.
Der Hauptteil der Aufgaben des Präprozessorsystems wird von so genannten Präprozessormodulen (PPM) ausgeführt. Jedes einzelne PPM kann 64 analoge Eingangssignale verarbeiten. Das gesamte System verwendet 128 identische PPMs, welche somit insgesamt bis zu 8192 Kanäle anbieten. Jeweils 16 PPMs werden in einem Elektronik-Crate untergebracht, zusammen mit 4 weiteren Kontrollmodulen. Ein Kontrollmodul beherbergt einen AMD Athlon Personal Computer, der als Kontroll- und Steuerungsinstanz für das gesamte Crate dient. Ein weiteres Modul empfängt das globale TTC-Signal und verteilt es an alle anderen Module des Crates. Wie im vorigen Kapitel erwähnt, übermittelt das TTC-Signal unter anderem das Level-1 Accept Signal. Die beiden letzten Kontrollmodule sind identisch und dienen dem Empfang und Weitertransport von Readout Daten von jeweils 8 PPMs (siehe Abbildung 7). Im Folgenden werden die einzelnen Einschübe dieses Crates näher erläutert.

[12] bunch-crossing – das Aufeinandertreffen zweier Protonenwolken (erfolgt alle 25 ns)
[13] LVDS – low-voltage differential signaling

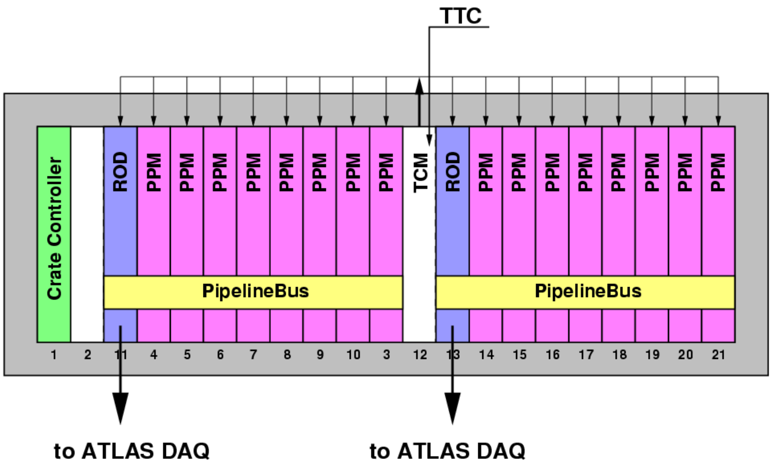


Abbildung 7 - Layout eines Präprozessor Crates

3.2.1 Die Struktur des Präprozessormoduls (PPM)

Ein Präprozessormodul besteht aus einem mehrschichtigen 9U[14] Printed-Circuit-Board (PCB) mit einer Vielzahl von so genannten Tochtermodulen. Ein Tochtermodul ist ein Multi-Chip-Modul (MCM), welches ebenfalls ein (kleines) PCB darstellt, auf welchem eine Anzahl von Chips in Die[15]-Form untergebracht sind. Diese Tochtermodule werden auf das große PPM aufgesteckt.

Die Verwendung von MCMs bringt den Vorteil größerer Flexibilität bei gleichzeitig einfacherer Herstellung. Sollte bei der Fertigung ein fehlerhafter Chip verwendet werden oder während des Betriebs ausfallen, so muss nur das eine betroffene MCM gegen ein funktionierendes Exemplar ausgetauscht werden anstatt des gesamten Präprozessormoduls. Auf diese Weise sinken der Ausschuss und damit die Kosten deutlich. Auch ist die Herstellung und Bestückung eines so großen PCBs durchaus nicht trivial, weshalb die Unterbringung fast aller vitalen Bauteile auf wesentlich kleineren Tochtermodulen sicherer ist.

Die einzelnen Komponenten des abgebildeten PPMs und ihre Funktion sollen im Weiteren kurz dargestellt werden (siehe auch Abbildung 8). Die für diese Diplomarbeit entscheidenden Komponenten werden im Anschluss noch einmal genauer erläutert:

[14] 9U: U ist hier ein Größenstandard für PCBs und Crates, 9U entsprechen 36,6cm

[15] die: ein roher Silikonchip ohne Verpackung oder Gehäuse (dadurch sehr klein)

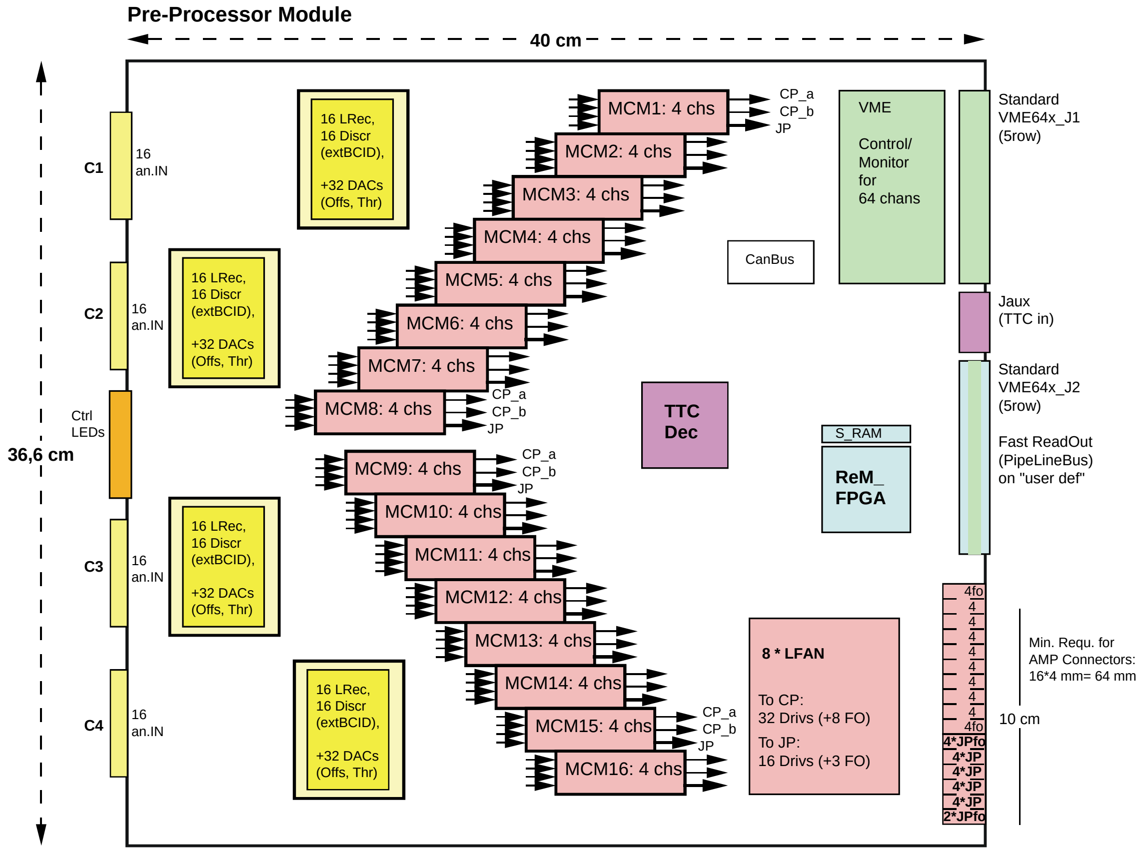


Abbildung 8 - Schema der Präprozessormoduls PPM

Das Analog-Input Modul (4 Stück pro PPM)

Dieses Modul nimmt jeweils 16 differentiale analoge Signale entgegen und wandelt sie in unipolare Signale um (gelbe Module in Abbildung 8). Dabei kann für jedes einzelne Signal eine individuelle Nullstellenkorrektur durchgeführt werden. Außerdem wird für jedes Signal bei Überschreiten einer ebenfalls individuell konfigurierbaren Schwelle ein digitales, so genanntes „externes BCID“ Signal erzeugt.
Für die Konfigurierung der Nullstellenkalibrierung und des BCID Schwellenwertes werden pro Kanal zwei DAC[16]s verwendet. Als DACs im Speziellen werden vier kommerzielle MAXIM MAX529 verwendet. Jeder MAX529 stellt acht 8Bit DACs zur Verfügung, die über die so genannte SPI[17]-Schnittstelle programmiert werden können.

Das Präprozessor Multi-Chip-Modul (16 PPrMCMs pro PPM)

Dieses Modul enthält die wichtigsten Komponenten des Präprozessors. Jedes PPrMCM verarbeitet vier Kanäle. Die vier analogen Eingangssignale werden von einem der Line-Receiver Module vorverarbeitet und danach direkt zu einem der 16 PPrMCMs weitergeleitet.
Das PPrMCM verrichtet angefangen bei der Digitalisierung der analogen Signale alle Aufgaben die in Kapitel 3.1 erwähnt werden.

[16] DAC – Digital-Analog-Konverter
[17] SPI – serielles Datenprotokoll mit 3 Leitungen, näheres sie Kapitel 3.3.2

Hierzu befinden sich auf dem PPrMCM mehrerer Chips in „Die“-Form:

- **4 Flash-ADCs** zur Digitalisierung der Eingangssignale
- **3 LVDS-Bausteine** zur differentiellen Weiterleitung der seriellen Ausgangssignale. Diese Daten sind die Eingangsdaten für den Cluster Prozessor (2 LVDS-Bausteine) und den Jet/Energy-Sum Prozessor (1 LVDS-Baustein) des Level-1 Triggers.
- **1 Phos4-Chip** zur Feinjustierung der Phase der LHC-Clock, welche als Digitalisierungssignal für die FADCs fungiert. Der Phos4-Chip wird über ein I²C-Interface[18] programmiert
- **1 PPrASIC** (Präprozessor ASIC). Der PPrASIC wurde am Kirchhoff Institut in Heidelberg entwickelt und erbringt die eigentliche Vorverarbeitungsleistung des Präprozessors. Er verarbeitet gleichzeitig die Signale von vier Kanälen. Der PPrASIC liefert auch die in Kapitel 3.1 erwähnten Readout Daten über zwei spezielle serielle Schnittstellen (siehe Kapitel 3.3.3).

Das LFAN Modul (1 Stück pro PPM)

Aufgabe dieser speziellen ASIC-Chips ist die Übertragung der von den LVDS-Ausgängen der PPrMCMs empfangenen Daten an die entsprechenden Module des Cluster Prozessors und des Jet/Energy-Sum Prozessors der Level-1 Triggers. Die Verbindung zu diesen Modulen erfolgt über Kabel mit einer Länge von bis zu 20 Metern. Die Signale müssen daher mit entsprechender Stärke getrieben werden, was für die kleinen LVDS-Chips auf dem PPrMCM nicht möglich ist.
Hierzu werden acht so genannte LFAN ASICs verwendet, welche in Packageform (nicht als „Die“) auf einem Tochtermodul platziert werden. Der LFAN ist ein speziell in Heidelberg entwickelter ASIC-Chip der jeweils 8 Ausgangskanäle mit einer Datenrate von bis zu 480 Megabyte pro Sekunde realisiert.
Von den PPrMCMs werden insgesamt 32 LVDS-Signale für den Cluster Prozessor und 16 LVDS-Signale für Jet/Energy-Sum Prozessor erzeugt. Allerdings müssen einige dieser Signale doppelt an zwei verschiedene CP oder JP Module gesendet werden, weshalb sechs LFANs nicht ausreichen und acht Chips verwendet werden müssen.
Die neueren Entwicklungen auf dem FPGA Sektor führen auch zu Überlegungen, eventuell statt relativ teuren LFAN-Chips neue günstige FPGAs einzusetzen, die ebenfalls in der Lage sind differentielle Signale mit der entsprechenden Stärke zu erzeugen. Während jedoch die Eignung der LFAN-Chips schon ausführlich getestet wurde [1], muss die Eignung der FPGAs noch ausgiebig analysiert werden.

Das TTCdec Modul (1 Stück pro PPM)

Das „Timing, Trigger and Control“ Modul (TTC-Modul, siehe Kapitel 3.2.2) empfängt über eine optische Verbindung Signale, die über die gesamte ATLAS Architektur verteilt werden. Diese optischen Signale werden vom TTC Modul empfangen und über die Backplane elektrisch an die anderen Crate-Module verteilt, wo sie von jeweils einem TTCdec Tochtermodul empfangen werden. Dazu von Bedeutung für das PPM ist unter anderem die so genannte LHC-Clock, welche die Protonen Bunch-Crossings taktet und das Level-1 Trigger Accept Signal. Die TTC Signale sind die einzige Möglichkeit Module und verschiedene Crates miteinander zu synchronisieren.
Dieses Tochtermodul wurde nicht speziell für den Präprozessor entwickelt, sondern es findet Verwendung in fast allen Modulen des ATLAS Experiments.

[18] I²C – serielles Datenprotokoll mit 2 Leitungen (Industriestandard), siehe Kapitel ???

Das CAN-Bus Modul (1 Stück pro PPM)

Das CAN-Bus Protokoll ist ein Industriestandard und wird im ATLAS Experiment für die Überwachung des Zustands der verschiedenen Crates, Module und Chips auf dem Modulen verwendet. Es werden Parameter wie Spannung, Stromaufnahme, Lüftergeschwindigkeit und Temperatur übermittelt. Dabei steht es jedem Modul frei, welche Parameter es über den CAN-Bus verfügbar macht.

Das VME-Bus Modul (1 Stück pro PPM)

Der VME-Bus ist ein Industriestandard zur asynchronen Datenübertragung. Jedes einzelne Modul der verschiedenen Präprozessor Crates kann über VME angesprochen werden. Der VME-Bus dient hierbei vor allem der Steuerung und Konfiguration der einzelnen Komponenten der verschiedenen Präprozessor Module. Jedes Crate-Modul besitzt eine eindeutige Adresse im Adressraum des VME-Busses.
Der VME-Bus des PPM wird von einem Xilinx CPLD implementiert. Das VME-Modul ist weiterhin für die Konfiguration des RemFPGA zuständig. Mit der Hilfe eines Flash-RAM kann das CPLD den RemFPGA mit einer von zwei möglichen Konfiguration laden (siehe auch das Kapitel über FPGA Technologien).
Das CPLD kümmert sich um alle Detailaspekte des komplizierten VME-Bus Protokolls und stellt dem RemFPGA einen stark vereinfachten Zugang zur VME-Kommunikation zur Verfügung.

Der Readout-Merger FPGA (1 Stück pro PPM)

Dieser FPGA, der Gegenstand dieser Diplomarbeit ist, befindet sich direkt auf dem PPM und nicht auf einem Tochtermodul. Es handelt sich um einen Xilinx Virtex-E 1000 in einem Ball-Grid-Array Package (BGA560) [2]. An der Virtex direkt angeschlossen ist ein 2 MB Speicherchip, er wird jedoch von dem in dieser Diplomarbeit vorgestellten Programmcode nicht benötigt. Er steht für spätere Erweiterungen zur Verfügung.
Der RemFPGA ist mit beinahe allen anderen Modulen des PPM verbunden und kontrolliert und steuert diese. Die Kommunikation mit der Außenwelt erfolgt einerseits über den VME-Bus (mit Hilfe des VME Moduls) und über den speziell entwickelten Pipeline-Bus, welcher vom RemFPGA direkt gesteuert wird.
Seine Hauptaufgaben bestehen in der Konfiguration des gesamten PPM und der Übertragung von Readout Daten über wahlweise den VME-Bus oder den Pipeline-Bus. Die genaue Funktion wird in den folgenden Kapiteln dargestellt.

3.2.2 Das TTC-Modul

Für die Synchronisation des gesamten ATLAS Experiments dienen eine Reihe von globalen ATLAS Signalen, die über optische Verbindungen an die einzelnen Komponenten des ATLAS Systems verteilt werden. Zum Empfang dieser Daten und der Konvertierung in elektrische Signale dient das TTC-Modul. Die empfangenen TTC-Signale werden an alle anderen Module des jeweiligen Crates verteilt. Jedes Crate des Level-1 Triggers verfügt über ein solches Modul.

Zu den Signalen gehören unter anderem:

- **LHC-Clock:** Dieses Signal taktet die einzelnen Bunch-Crossings des LHC-Beschleunigers. Dieses Signal ist für die Synchronisation des gesamten ATLAS Systems unerlässlich.
- **Level-1 Kalorimeter Accept:** Dieses Signal zeigt eine positive Entscheidung des Level-1 Kalorimeter Triggers an. Viele Triggerkomponenten erzeugen daraufhin Readout Daten zur Kontrolle der Performance des Triggers.

- **Reset der BC-Nummer:** Bei jedem LHC-Clock Signal inkrementieren Komponenten wie zum Beispiel der PPrASIC einen internen Zähler. Mit Hilfe dieses Signals wird die so genannte BC-Nummer zurückgesetzt und so gewährleistet, dass alle Komponenten des ATLAS Systems global und synchron mit derselben Nummer arbeiten.
- **Reset der Level-1 Event-Nummer:** Ebenso wie für die obige BC-Nummer existiert auch ein Reset-Signal für die so genannte Event-Nummer. Jede positive Level-1 Trigger Entscheidung inkrementiert diese Zahl.

3.2.3 Der Crate-Controller des Präprozessorsystems

Bei diesem Crate-Einschub handelt es sich um einen am Kirchhoff Institut in Heidelberg entwickelten AMD Athlon PC mit angeschlossenem VME-Interface. Der PC verfügt über eine Festplatte einen Ethernet Netzwerk Anschluss.
Sämtlich Kommunikation über den VME-Bus wird von diesem Crate-Controller gesteuert.

3.2.4 Der Readout-Driver des Präprozessorsystems

Für jedes Präprozessor-Crate werden zwei Readout-Driver Module (ROD) verwendet. Jeweils acht PPMs sind über den am Kirchhoff Institut entwickelten Pipeline-Bus mit jeweils einem ROD verbunden. Der ROD verfügt darüber hinaus über einen Anschluss an den VME-Bus. Jedoch wird dieser nicht für die Kommunikation zwischen ROD und PPM verwendet, sondern ausschließlich für Datenverkehr zwischen Crate-Controller und ROD.
Die von den verschiedenen PPrASICs generierten Readout-Daten werden vom ROD Modul empfangen und über eine Anzahl von so genannten S-LINK Verbindungen an das Datenverarbeitende System des ATLAS Level-1 Kalorimeter Triggers weitergeleitet. S-LINK Verbindungen existieren in verschiedenen Geschwindigkeitsausführungen und die genaue Zahl und Art der zu verwendenden Verbindungen war zum Zeitpunkt dieser Diplomarbeit noch nicht endgültig festgelegt.
Der Großteil der Funktionalität des ROD wird ebenfalls von einem Xilinx Virtex-E 1000 implementiert werden. Zum Zeitpunkt dieser Arbeit existiert lediglich eine ältere Version des Programmcodes des ROD-FPGAs, welcher für einen Prototyp des Readout-Mergers entwickelt wurde. Dieser Prototyp war in einem ASIC implementiert (RemASIC).
Der neue RemFPGA, der im Rahmen dieser Arbeit entwickelt wurde ist mit dem Interface des RemASIC nicht mehr kompatibel. Der noch zu erstellende neue Code des ROD-FPGA kann jedoch voraussichtlich zu weiten Teilen von dem bestehenden Programmcode abgeleitet werden.

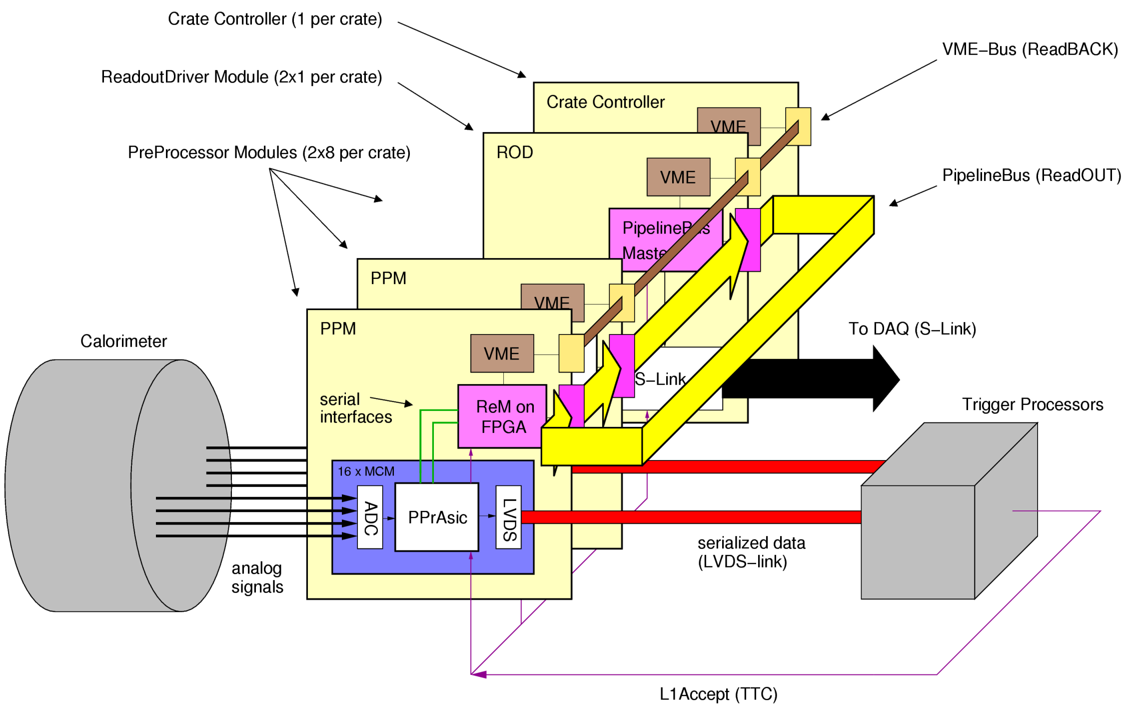


Abbildung 9 - Verbindungen der einzelnen Crate-Module (TTC nicht dargestellt)

3.3 Schnittstellen des RemFPGA

Im Folgenden sollen die Strukturen und Protokolle der verschiedenen Schnittstellen erläutert werden, die mit dem RemFPGA verbunden sind.
Dabei ist grundsätzlich zu unterscheiden zwischen Protokollen, die von verwendeten anderen Chips (z.B. Phos4 oder PPrASIC) festgelegt werden und Schnittstellen deren Implementierung teilweise oder ganz von der Programmierung des RemFPGA bestimmt werden.

3.3.1 Das I²C Protokoll

Dieses Protokoll ist ein von Philips entwickelter Industriestandard [3]. Das I²C Protokoll benötigt lediglich zwei Leitungen und ermöglicht die Kommunikation von einer großen Anzahl von Slaves mit mehreren Mastern.
Ein *Master* ist hierbei definiert als ein angeschlossenes beliebiges Gerät (ASIC, FPGA, Mikroprozessor), welches einen Datentransfer aktiv einleitet. Ein *Slave* ist eine Instanz, die sich auf dem I²C-Bus passiv verhält bis sie von einem Master zur Kommunikation aufgefordert wird. Dabei ist jedem Slave je nach verwendetem optionalem Zusatzprotokoll eine eindeutige Adresse von 7 oder 10 Bit Länge zugeordnet. Die Datenübermittlung erfolgt in Bytes von 8 Bit Länge.
Das I²C-Protokoll ist definiert für drei verschiedene Geschwindigkeitsstufen von 100kBit (Normal), 400kBit (Fast) und 3,4MBit (Highspeed) pro Sekunde. Dabei ist der 400kBit Modus abwärtskompatibel zum Normalmodus, während der Highspeed-Betrieb einer hierzu inkompatiblen Spezifikation folgt.
Im Falle der PPM wird lediglich der 100kBit-Modus verwendet und der RemFPGA fungiert als einziger Master. Daher werden im Folgenden nur die vom PPM benötigten I²C-Funktionalitäten kurz erläutert. Für eine ausführlichere Darstellung sei auf die Spezifikation von Philips verwiesen [4].

Die I²C-Datenleitungen und Master-Server Kommunikation

Die beiden Leitungen des I²C-Busses bestehen aus bidirektionalen Leitungen, die an einen Pull-Up Widerstand angeschlossen sind. Wenn kein Gerät den I²C-Bus anspricht, ist der Zustand der beiden Leitungen somit HIGH.
Eine Leitung fungiert als Clocksignal (*SCL*) und die zweite Leitung als Datensignal (*SDA*).
Bei dem Transfer von Datenbits muss der Zustand von SDA während der HIGH-Periode von SCL stabil sein. Für jedes zu übertragende Bit wird vom I²C-Master ein Clocksignal generiert (siehe Abbildung 10).

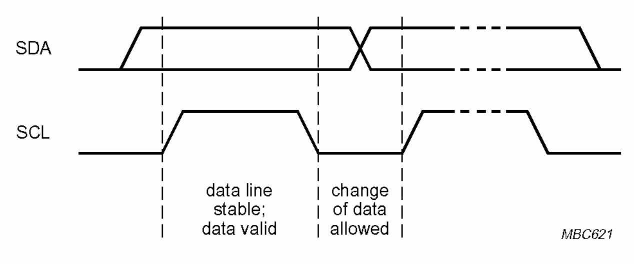


Abbildung 10 - Datentransfer auf dem I²C-Bus

Zusätzlich definiert der I²C-Bus die Generierung zweier spezieller Befehlssignale namens START und STOP. Diese werden durch einen HIGH-LOW (START) bzw. einen LOW-HIGH (STOP) Übergang auf SDA generiert während SCL HIGH ist (siehe Abbildung 11).

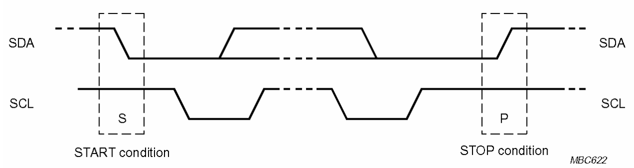


Abbildung 11 - START und STOP Bedingungen

Der Master initiiert einen Datentransfer durch das Generieren eines START-Signals. Hierzu zieht der Master SDA auf LOW während SCL HIGH ist. Danach wird SCL ebenfalls auf LOW gezogen.
Benötigt ein an den I²C-Bus angeschlossenes Gerät Zeit, um auf Aktivität auf dem Bus zu reagieren, so kann es ein vom Master auf LOW gezogenes SCL-Signal ebenfalls aktiv auf LOW ziehen. Dadurch bleibt SCL LOW, selbst wenn der Master die Leitung wieder freigibt. Der Master erkennt dies und wartet auf eine Freigabe des SCL Leitung durch das andere Gerät. Auf diese Weise kann jedes I²C-Gerät einen Wartezustand auf dem Bus erzwingen. Bei einer nicht ordnungsgemäßen Funktion eines angeschlossenen Geräts kann dies andererseits aber auch den kompletten Bus dauerhaft blockieren.
Nach der START-Bedingung sendet der Master 8 Bits auf den Bus und generiert anschließend einen neunten Clockimpuls auf SCL. SDA wird während dieses neunten Impulses jedoch nicht vom Master getrieben. Stattdessen ist es die Aufgabe des angesprochenen Slave-Gerätes, die SDA-Leitung auf LOW zu ziehen um den Empfang zu bestätigen, im I²C Jargon „Acknowledgement“ genannt (siehe Abbildung 12).

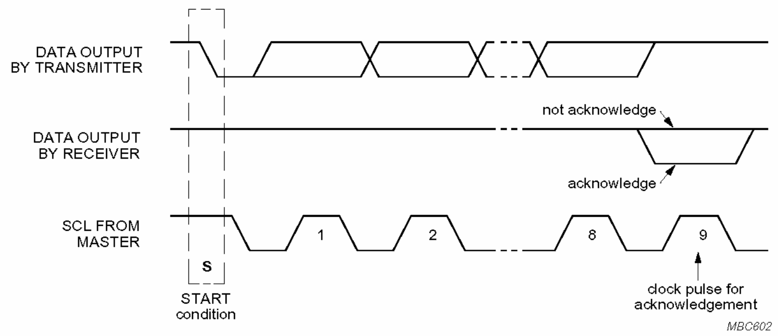


Abbildung 12 - I²C Acknowlegement

Die übermittelten 8 Bits des Masters enthalten die Adresse des angesprochenen Slaves und die Richtung für den darauf folgenden Datentransfer.
Im Anschluss generiert der Master weitere 9 Impuls auf SCL. Je nach Datenrichtung sendet oder empfängt der Master hierbei Daten von bzw. zu dem angesprochenen Slave. Der neunte Clockimpuls dient wieder der Bestätigung des Empfangs – falls der Master Daten von einem Slave ausgelesen hat wird dieses Acknowledgement-Signal vom Master generiert. Beliebig viele weitere 9Bit-Zyklen schließen sich optional an.
Um den Datentransfer zu beenden, sendet der Master ein STOP-Signal nach einer 9Bit Datenübertragung. Alle Geräte müssen daraufhin auf dem I²C-Bus zurück in den Bereitschaftszustand gehen. Um statt dessen einen weiteren Datentransfer anderer Richtung oder Adresse einzuleiten, sendet der Master statt des STOP-Signals ein neues START-Signal mit nachfolgend gesendeter Adresse des neuen Slaves. Siehe auch Abbildung 13 und Abbildung 14.

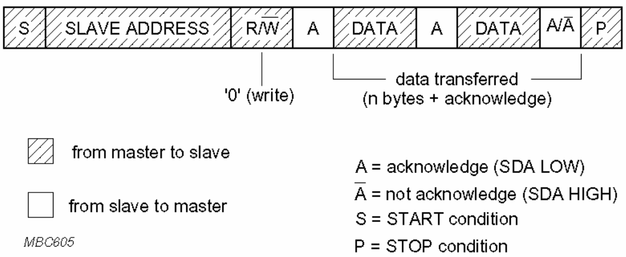


Abbildung 13 - I²C Datentransfer Master zu Slave

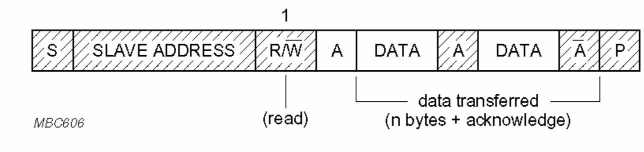


Abbildung 14 - I²C Datentransfer Slave zu Master

I²C Adressierungsschema und Besonderheiten des Phos4 Chips

Einige Adressen des I²C-Busses haben spezielle Bedeutungen, sie sind im Folgenden ohne weitere Erläuterungen aufgeführt:

Adresse	R/W Bit	Bedeutung
0000 000	0	„General-Call“
0000 000	1	START byte
0000 001	x	CBUS Adresse

0000 010	x	Reserviert für andere Busformate
0000 011	x	Reserviert für zukünftige Erweiterungen
0000 1xx	x	verwendet von High-Speed I²C
1111 1xx	x	Reserviert für zukünktige Erweiterungen
1111 0xx	x	10 Bit Slave Adresse

Tabelle 1 - I²C Sonderadressen

Einer „General-Call“ Adresse folgt ein weiteres Byte mit spezieller Bedeutung. Erwähnenswert ist hier der Wert 00000110, welcher einen Reset an alle angeschlossenen Geräte signalisiert. Die Beachtung dieses Resetsignals ist jedoch für jedes I²C-Gerät optional.

Der Phos4 Chip [5] besitzt einige Besonderheiten in seiner Implementierung des I²C-Busses. Falls in zukünftigen Versionen des PPMs andere Chips außer dem Phos4 an I²C angeschlossen werden sollen, müssen einige Einschränkungen beachtet werden.
Der Phos4 Chip besitzt nur vier Pins um die I²C-Adresse zu definieren, die restlichen drei Bits sind teilweise statisch definiert und werden teilweise ignoriert. Darüber hinaus kann der Chip nur beschrieben, aber nicht ausgelesen werden.

Adressbit	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	**A0**	**R/W**
Bedeutung	0	Pin20	Pin21	Pin22	Pin23	x	x	0

Tabelle 2 - Adressschema des Phos4

Zusammen mit Tabelle 1 ergibt sich somit zwingend die Einschränkung, dass für jeden Phos4 zumindest eines der drei Bit A5, A4 und A3 gesetzt sein muss, da der Chip ansonsten auch auf einige Sonderadressen reagieren würde. Es empfiehlt sich auch eventuelle weitere I²C-Geräte nur mit Adressen mit gesetztem A6-Bit zu versehen. Werden diese beiden zusätzlichen Randbedingungen beachtet, so bleibt der I²C-Standard weiterhin gewahrt.

3.3.2 Das SPI-Protokoll

Bei dem SPI-Protokoll handelt es sich um ein äußerst einfaches Protokoll, welches drei Datenleitungen verwendet. Die SPI-Schnittstelle wird ausschließlich von den DAC-Chips MAX529 von Maxim [6] auf den Analog-Input Modulen verwendet.
Es handelt sich um ein serielles Schiebeprotokoll, bei dem mehrere MAX529 optional hintereinander geschaltet werden können. Im Falle des Einsatzes auf den Analog-Input Modulen werden jeweils vier MAX529 zusammen geschaltet (daisy-chaining). Die drei Leitungen übertragen das Datensignal DIN, die Clock CLK und das Ladesignal CS. Jeder MAX529 verfügt zusätzlich über einen Pin DOUT, der mit dem DIN Eingang des nächsten MAX529 verbunden wird.
Jeder MAX529 implementiert ein 16 Bit Schieberegister. Bei jedem Clocksignal wird der Wert von DIN in das unterste Bit des Schieberegisters geschoben. Das oberste Bit wird auf DOUT ausgeben und dient so als Eingangssignal für den nächsten MAX529 in der Kette.
Bei einer Kettenlänge von Vier kann jedes Bit in jedem MAX529 auf einen eindeutigen Wert gesetzt werden, indem nacheinander 64 Bits auf dem SPI-Bus ausgegeben werden. Ein nachfolgendes Ladesignal CS veranlasst alle MAX529 den momentanen Inhalt ihrer Schieberegister als neue Programmierung zu übernehmen.
Die Geschwindigkeit des SPI-Busses beträgt im Falle des PPM 1 MHz.

3.3.3 Die serielle Schnittstelle des PPrASIC

Jeder PPrASIC verfügt über zwei identische serielle Schnittstellen. Jede steuert zwei der vier Kanäle der PPrASIC. Somit sind auf dem gesamten PPM 32 serielle Schnittstellen vorhanden. Diese werden mit einer Geschwindigkeit von 40MHz betrieben, wobei das Taktsignal vom RemFPGA zu erzeugen ist.
Das Protokoll ist spezifisch für den PPrASIC und verwendet vier Datenleitungen namens Clock, Frame, DataIn und DataOut. Die Bitlänge beträgt 13 Bit. Durch die getrennten Leitungen für Eingabe und Ausgabe können verschiedene Daten in beide Richtungen gleichzeitig übertragen werden. Der Beginn eines neuen Datenwortes wird dabei durch das Frame-Signal angezeigt, welches somit zusammen mit jedem dreizehnten Clocksignal erzeugt wird. Hier soll im Folgenden lediglich das Datenformat der übertragenen 13 Bit Worte kurz erläutert werden.
Für eine eingehendere Beschreibung der seriellen Schnittstelle siehe das Benutzerhandbuch des PPrASICs [7] und die Diplomarbeit zum PPrASIC [12].

Eingabewörter des PPrASIC

Über die serielle Schnittstelle können sowohl Befehle wie auch Daten für die internen Speicherbereiche und Register an den PPrASIC übermittelt werden. Der Wert des höchsten Bits entscheidet dabei, um welche Art von Eingabe es sich handelt. Der RemFPGA dieser Diplomarbeit erzeugt und sendet keine eigenen Befehle oder Daten über die serielle Schnittstelle zum PPrASIC. Es werden lediglich Datenblöcke über die VME-Schnittstelle (siehe Abschnitt 3.3.4) oder den Pipelinebus (siehe Abschnitt 3.3.5) entgegen genommen und unverändert an die angeschlossenen PPrASICs geschickt (siehe Abschnitt 6.3).

Ausgabewörter des PPrASIC

Abhängig vom Wert des höchsten Bits überträgt der PPrASIC ReadOUT- oder ReadBACK-Daten. Als ReadOUT bezeichnet man Daten, die zur Überprüfung des Verhaltens des Triggers bei jedem Level-1 Accept generiert werden und die über die beiden ROD (Readout Treiber) weitergeleitet werden. Als ReadBACK werden Daten aus den internen Registern und Speicherbereichen des PPrASIC bezeichnet.
Wird zum Beispiel der PPrASIC mit Hilfe des Eingabe-Befehls *StartReadback 0010000100011* zur Auslese des Registers Nummer 2 aufgefordert, so sendet der PPrASIC den Registerinhalt als ReadBACK Daten.
ReadOUT-Daten dagegen setzen sich aus so genannten RAW- und BCID-Werten in Verbindung mit einem vorangehenden Header zusammen. Ein solches Paket aus Header-, BCID- und RAW-ReadOUT-Datenwörtern wird von dem PPrASIC bei jedem Level-1 Accept für jeden der vier Kanäle generiert und ohne Aufforderung über die beiden seriellen Schnittstellen ausgegeben.
Bei den RAW-Werten handelt es sich um die unverarbeiteten Rohdaten, die von dem PPrASIC als 10 Bit Worte von den FADCs empfangen werden. Aus jedem RAW-Wert wird über mehrere Algorithmen ein 8 Bit BCID-Wert abgeleitet. Diese BCID-Daten sind das Endresultat des PPrASIC und werden auch (in leicht veränderter Form) über den so genannten Echtzeitdatenpfad an den Cluster Prozessor und den Jet/Energy-Sum Prozessor des Level-1 Triggers weitergeleitet. Die einzelnen BCID-Werte werden dabei nicht nur von einem RAW-Wert abgeleitet, sondern sind auch abhängig von vorangegangenen RAW-Daten.
ReadOUT-Daten und ReadBACK-Daten werden in einer unveränderlich definierten Reihenfolge gesendet (siehe Abbildung 15). Zuerst wird ein einzelnes ReadBACK-Wort gesendet. Dieses wird gefolgt von einem EventHeader-Wort für den ersten Kanal, welches weiter gefolgt wird von einem oder mehreren BCID-Werten (bis zu 31 Stück, Standardeinstellung: 1 BCID-Wort). Nach dem/den BCID-Werten folgen 0 bis 7 RAW-Werte (Standardeinstellung: 5 RAW-Werte). Anschließend folgen die ReadOUT-Daten des zweiten Kanals.

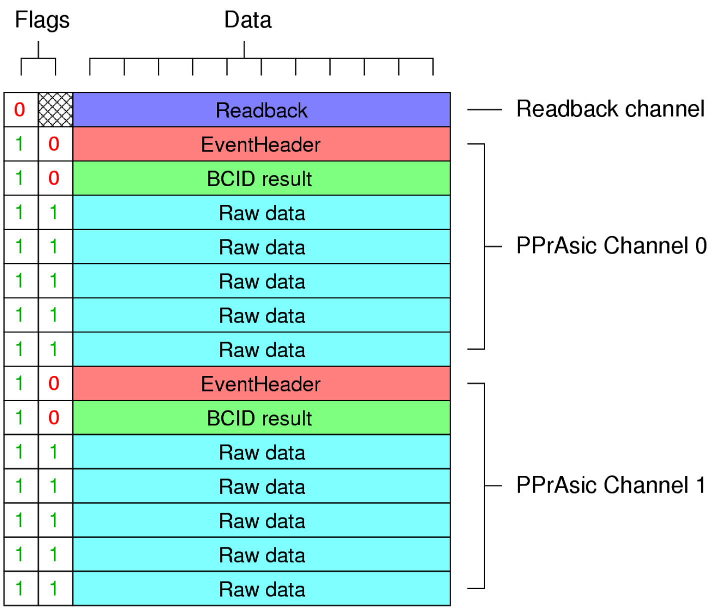


Abbildung 15 - Abfolge der seriellen Datenausgabe

Die Anzahl zu sendender BCID- und RAW-Werte kann für jeden Kanal einzeln über interne Register des PPrASIC eingestellt werden. In der Grundeinstellung werden für jeden Kanal 1 BCID- und 5 RAW-Werte übertragen. Ein RAW-Wort besteht aus einem 10 Bit-Wert und einem Flagbit namens „External BCID“. Ein BCID-Wort besteht aus einem 8 Bit Wert und 3 Flagbits. Jedes Header-Wort enthält die jeweils unteren 4 Bits des PPrASIC-internen Event- und Bunch-Crossing-Zählers. Bei einer ordnungsgemäßen Synchronisation des Präprozessorsystems sind diese Werte für alle 7296 Kanäle identisch. Für eine nähere Beschreibung der Bedeutung der einzelnen Daten siehe die Spezifikation des PPrASIC [7].

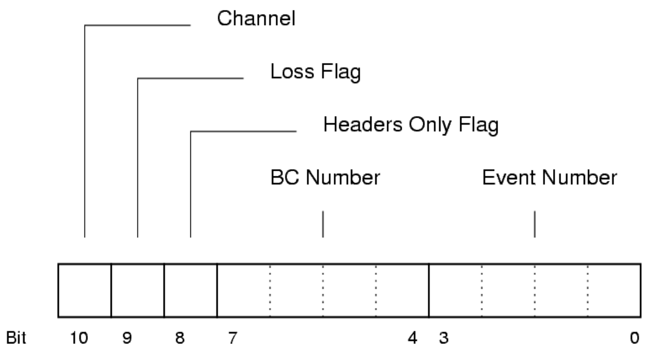


Abbildung 16 - serieller ReadOUT Header

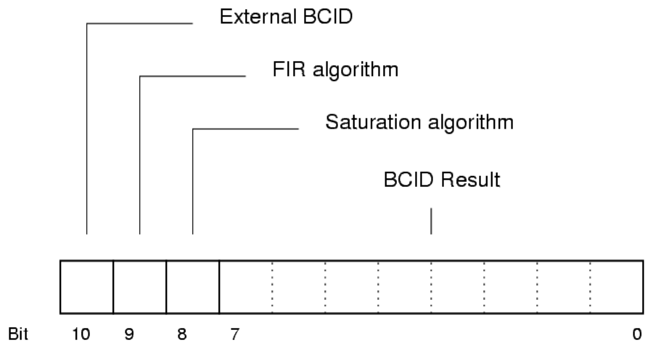


Abbildung 17 - serieller ReadOUT BCID-Wert

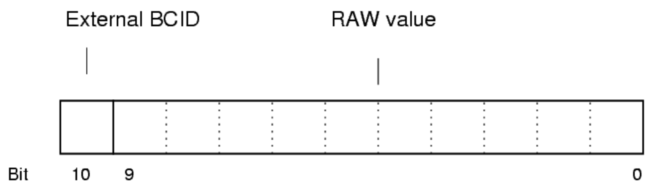


Abbildung 18 - serieller ReadOUT RAW-Wert

Falls mangels eines Level-1 Accept Signals keine ReadOUT-Daten gesendet werden sollten, werden einfach nur ReadBACK-Worte direkt nacheinander gesendet. Falls keine ReadBACK-Daten zu senden sind (unabhängig von eventuell zwischen einzelnen ReadBACK-Worten eingefügten ReadOUT-Worten), wird als ReadBACK fortlaufend ein so genanntes ReadBACK-Statuswort gesendet. Dieses gibt über Flagbits Auskunft über einige grundlegende Zustände des PPrASICs.

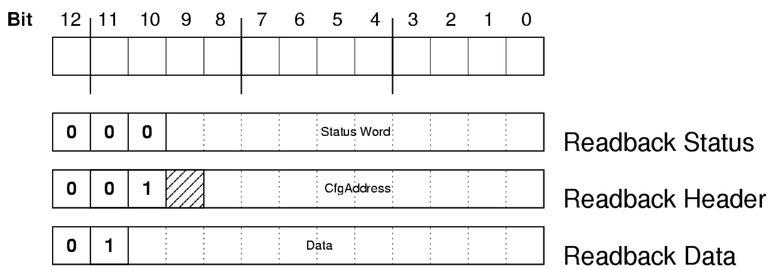


Abbildung 19 - serielle ReadBACK Daten

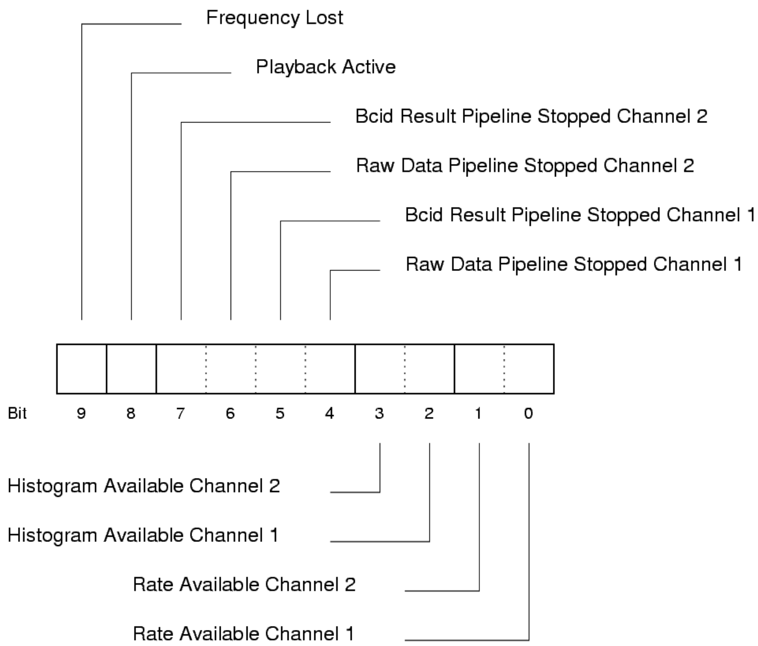


Abbildung 20 - serielles ReadBACK Statuswort

Die genaue Bedeutung der verschiedenen Bits dieser seriellen Datenwörter wird in [12] und in [7] eingehend erläutert.

3.3.4 Die Schnittstelle zum VME-CPLD

Die eigentliche Implementation der VME-Schnittstelle erfolgt durch das CPLD des VME-Moduls. Die über VME zu übermittelnden Daten werden allerdings vom RemFPGA generiert bzw. entgegengenommen. Die Kommunikation zwischen CPLD und RemFPGA erfolgt über eine VME-CPLD spezifische Schnittstelle namens „Easy VME“. Darüber hinaus liefert das CPLD auch das Clocksignal und das Resetsignal für den RemFPGA.
Der VME-Bus wird im Fall des PPM mit einer Datenbreite von 32 Bit und einer Adresstiefe von 22 Bit betrieben. Hinzu kommen eine Reihe von Steuersignalen die im folgenden aus der Sicht des RemFPGA näher beschrieben werden sollen (siehe Tabelle 1). Alle Steuersignale sind grundsätzlich low-aktiv.

Signal	Richtung	Bedeutung
Interrupt	out	Programmierbares Interruptsignal des RemFPGA
Select 1	in	Ein gültiger Wert liegt an der Adressleitung an
Select 2	in	reserviert für zukünftige Verwendung
R/W	in	Schreib/Lese-Zugriff
DS	in	Ein gültiger Wert liegt an der Datenleitung an (bei Schreiboperation)
RDY	out	Ein gültiger Wert liegt an der Datenleitung an (bei Leseoperation)

Tabelle 3 - Steuersignale von "Easy VME"

Eine **Leseoperation** funktioniert schrittweise wie folgt:

- VME-CPLD legt einen 22 Bit Wert auf die Adressleitung
- VME-CPLD setzt R/W HIGH (Leseoperation)
- VME-CPLD zieht Select 1 auf LOW
- RemFPGA verarbeitet die Anfrage
- RemFPGA legt einen 32 Bit Wert auf die Datenleitung
- RemFPGA zieht RDY auf LOW
- VME-CPLD verarbeitet die Eingabe
- VME-CPLD setzt Select 1 zurück auf HIGH
- RemFPGA setzt RDY zurück auf HIGH

Eine **Schreiboperation** funktioniert schrittweise wie folgt:

- VME-CPLD legt einen 22 Bit Wert auf die Adressleitung
- VME-CPLD legt einen 32 Bit Wert auf die Datenleitung
- VME-CPLD zieht R/W auf LOW (Schreiboperation)
- VME-CPLD zieht DS auf LOW (Daten stehen bereit)
- VME-CPLD zieht Select 1 auf LOW
- RemFPGA verarbeitet die Eingabe
- RemFPGA zieht RDY auf LOW
- VME-CPLD setzt Select 1 zurück auf HIGH
- RemFPGA setzt RDY zurück auf HIGH

Sämtliche Signale sind vollständig asynchron. Das Interruptsignal des RemFPGA wird über interne Register des FPGA gesteuert. Eine Beschreibung der Adresszuordnung der VME Schnittstelle folgt in Kapitel 6.8.

3.3.5 Das Pipelinebus Protokoll des RemFPGA

Der Pipelinebus wurde speziell für den Transport von ReadOUT Daten des Präprozessors entwickelt. Er vereint eine hohe Durchsatzrate bei geringer Störanfälligkeit mit einer relativ einfach zu implementierenden Schnittstelle und speziell auf den Bedarf des Präprozessors zugeschnittene Steuersignale.
Der Bus verwendet einen 32 Bit Datenbus zuzüglich zweier Steuersignale, einem Paritysignal und einem Clocksignal. Für diese Signale existieren jeweils getrennte Eingangs- und Ausgangsleitungen. Entsprechend Abbildung 9 (Seite 24) ist jede Pipelinebus Einheit (im weiteren *Node* genannt) jeweils mit der vorigen Node und der nachfolgenden Node verbunden. Dabei ist jeweils der Pipelinebus-Ausgang einer Node mit dem Eingang der nächsten Node verbunden bis ein geschlossener Ring entsteht. Dieser Ring ist unidirektional, durch die geschlossene Ringform kann allerdings trotzdem jede Node mit jeder anderen Node kommunizieren. Die Pipelinebus Nodes sind also nicht parallel auf den Bus aufgeschaltet was einen wesentlichen Beitrag zu Störsicherheit des Systems leistet.
Bei jedem Clocksignal wird das Eingangssignal in ein internes Latch-Register übertragen, während der vorige Inhalt des Registers auf den Ausgang gestellt wird. Somit wandert ein Datenwort mit jedem Clocksignal von einer Node weiter zur nächsten Node (Abbildung 21).

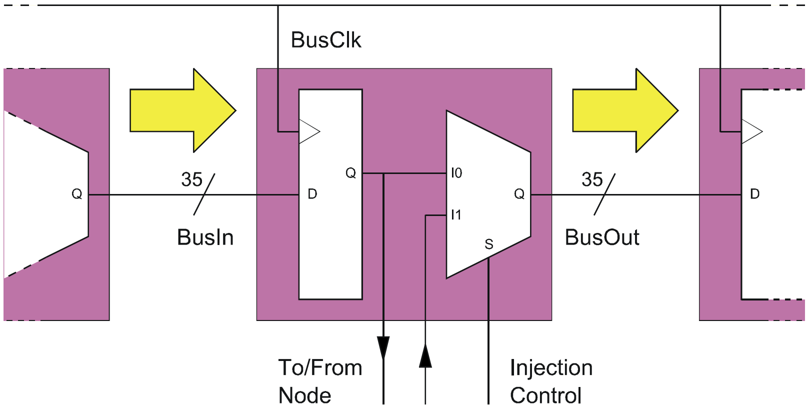


Abbildung 21 – Latch-Register des Pipelinebusses

Jeweils 8 PPMs sind mit einem ROD verbunden. Dabei agiert der ROD als Master, welcher als einziger neue Datenwörter auf den Bus ausgibt und sie nach einem kompletten Zyklus durch den Ring am Eingang wieder entgegen nimmt (und im Normalfall vom Bus entfernt). Somit steht neben dem Paritysignal eine weitere Möglichkeit zur Fehlererkennung zur Verfügung: vom ROD auf dem Pipelinebus ausgegebene Daten müssen nach 9 Clockzyklen wieder am Eingang des ROD ankommen.
In diesem Abschnitt soll lediglich die Grundstruktur des Pipelinebusses erläutert werden. Die Details des Pipelinebusses werden in einem folgenden Kapitel ausführlich dargelegt.

Struktur der Pipelinebus Daten

Die beiden Kontrollleitungen des Pipelinebusses ermöglichen die Unterscheidung von verschiedenen Typen von 32 Bit Datenwörtern die über den Bus übertragen werden. Es wird zwischen *Befehlswörtern (Command words)*, *Datenwörtern (Data words)* und *leeren Wörtern (Empty words)* unterschieden. Zwei Kontrollsignale ermöglichen insgesamt vier Typen, der Pipelinebus definiert jedoch nur drei und der verbleibende Typ ist reserviert für zukünftige Erweiterungen des Protokolls (siehe Abbildung 22 und Abbildung 23).

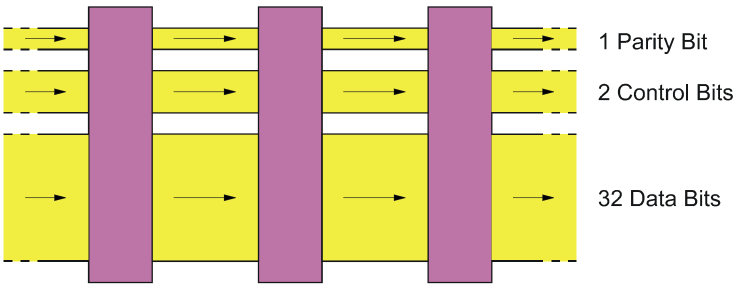


Abbildung 22 - Datenbreite des Pipelinebusses

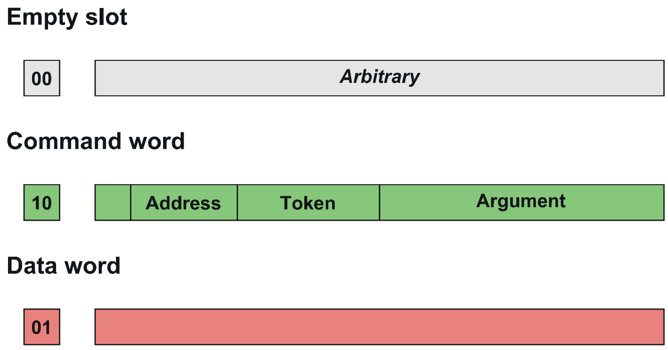


Abbildung 23 - Pipelinebus Wörter

Ein *Befehlswort* besteht aus einer 6 Bit Node-Adresse, einem 8 Bit Befehlscode, einem 16 Bit Befehlsargument und zwei Flagbits.
Jedem Pipelinebus Slave sind jeweils eine eindeutige Node-Adresse und eine Gruppen-Adresse zugeordnet. Darüber hinaus ist noch eine Broadcast-Adresse definiert, welche jeden Slave auf dem Bus anspricht. Die verschiedenen Pipelinebus Befehle werden durch den 8 Bit Code identifiziert und jedem Befehl steht das 16 Bit Argument zur beliebigen Verwendung zur Verfügung.
Empfängt eine Slave-Node ein an sie gerichtetes Befehlswort und erkennt und verarbeitet dieses als solches, so setzt sie das Accept-Bit auf 1 und schickt den damit „akzeptierten“ Befehl weiter auf seinem Weg durch den Ring zurück zum Bus-Master. Bei fehlerfreier Funktion des Pipelinebusses darf niemals ein Befehlswort ohne gesetztes Accept-Bit zurück bei Master eintreffen.
Erkennt ein Slave einen Fehler auf dem Pipelinebus wie zum Beispiel einen Parityfehler oder ein ungültigen Befehlscode, so ersetzt dieser Slave das fehlerhafte Wort (welches zum Beispiel auch ein Datenwort mit fehlerhafter Parity sein kann) durch ein Befehlswort mit gesetztem Error-Bit. In diesem Fall enthält der Befehlscode eine Beschreibung des aufgetretenen Fehlers (siehe Kaptitel 6.7).

Datenübertragung über den Pipelinebus

Wie zuvor erwähnt ist es nur dem Pipelinebus Master erlaubt, Wörter auf dem Bus auszugeben. Der Master initiiert und kontrolliert somit jeglichen Datenverkehr über den Bus. Eine Slave-Node kann nicht aus eigener Initiative Gebrauch von dem Pipelinebus machen, sondern muss auf die Initiierung einer Kommunikation durch den Master warten. Allerdings muss es für einen Slave natürlich eine Möglichkeit nicht nur zum Empfang, sondern auch zur Ausgabe von Daten geben. Zu diesem Zweck ist es einem Slave erlaubt *leere Wörter* durch *Datenwörter* zu ersetzen. Im Folgenden wird das Ausführen einer Lese- und einer Schreiboperation über den Pipelinebus kurz näher erläutert.

Schreiboperation – Datenübertragung von Master zu Slave(s)

Zuerst gibt der Master ein spezielles Befehlswort auf den Bus, den so genannten *BeginOfData* Befehl. Seine Adresse bestimmt den oder die Slaves, die die folgenden Daten entgegennehmen sollen. Mehrere Slaves können durch die Verwendung einer Gruppenadresse oder der Broadcast-Adresse angesteuert werden. Die so adressierten Slaves gehen daraufhin in Empfangsbereitschaft.
Der BeginOfData Befehl wird direkt gefolgt von einer beliebigen Anzahl von Datenwörtern, welche von den entsprechenden Nodes eingelesen werden und unverändert über den restlichen Bus weitergeleitet werden.
Der Datenstrom wird abgeschlossen durch einen *EndOfData* Befehl. Grundsätzlich müssen Datenwörter immer von umgebenden BeginOfData und EndOfData Befehlen eingeschlossen sein.

Leseoperation – Datenübertragung von Slave(s) zu Master
Auch hier beginnt der Master wieder mit einem entsprechend adressierten *BeginOfData* Befehl. Es handelt sich hierbei um exakt das gleiche Befehlswort wie für die Initiierung einer Schreiboperation (das Befehlsargument wird nicht verwendet). Dies bedeutet, dass die Datenrichtung durch zusätzliche Befehle gesteuert werden muss, ebenso wie die Auswahl der Art und Bedeutung der zu übertragenden Daten. Diese Befehle werden in der Referenz des Pipelinebusses (siehe Kapitel 6.7) erläutert. Der oder die angesprochenen Slaves gehen daraufhin in Ausgabebereitschaft.
Der BeginOfData Befehl wird gefolgt von *leeren Wörtern*. Der erste adressierte Slave ersetzt diese leeren Wörter durch die von ihm auszugebenden Datenwörter. Der Slave schließt seine Übertragung ab durch ein von ihm erzeugten *EndOfData* Befehl. Die 6 Bit Adresse dieses Befehls enthält die diesem Slave zugeordnete eindeutige Adresse, welche somit den Ursprung des vorigen Datenblocks identifiziert. Nach dem Abschluss der Ausgabe geht der Slave zurück in den Ruhezustand.
Sollte der Master mehr als einen Slave angesteuert haben, so warten auch diese Slaves auf leere Wörter, die sie mit Datenwörtern ersetzten können. Da jedoch der erste Slave die direkt auf den BeginOfData Befehl folgenden leeren Wörter durch Datenwörter ersetzt hat, müssen die folgenden Slaves auf das Ende dieser Daten warten. Dies geschieht automatisch – jeder Slave hat einfach auf leere Wörter zu warten, welche in diesem Fall nach dem vom ersten Slave ausgegeben EndOfData folgen. Direkt hinter EndOfData fügt also der zweite angesprochene Slave seine Daten ein und schließt diese wiederum durch ein eigenes EndOfData ab (siehe Abbildung 24).

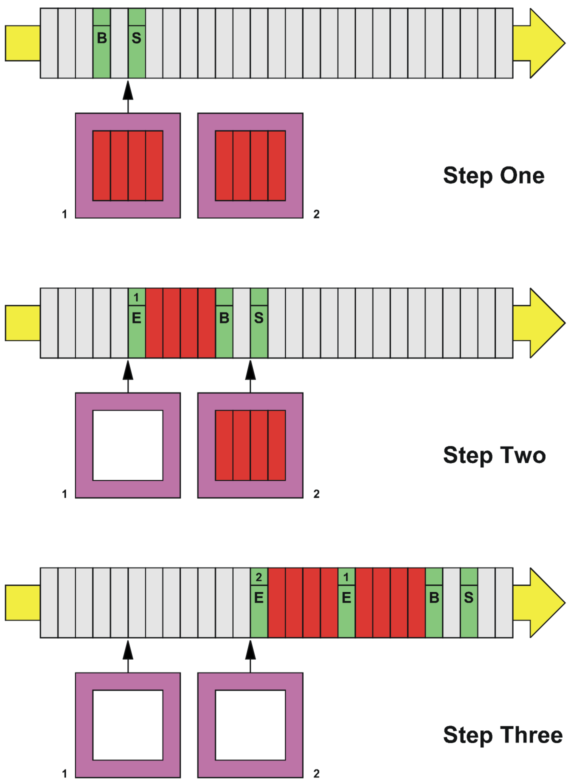


Abbildung 24 - Datentransfer vom Slave zum Master

Auf diese Weise kommen die verschiedenen angesprochenen Slaves nacheinander an die Reihe. Sollte der Master alle acht Slave PPM angesteuert haben, so empfängt er auf seinem Eingang nach 9 Zyklen zuerst sein eigenes BeginOfData gefolgt von acht Datenblöcken, die jeweils durch ein EndOfData abgeschlossen werden. Danach empfängt der Master unveränderte leere Wörter so wie er sie auf dem Ausgang erzeugt hat. Erst jetzt wenn der Master nach dem Empfang des kompletten Datenpakets wieder seine eigenen leeren Wörter empfängt, kann er statt dieser leeren Wörter wieder andere Befehle oder Daten auf den Pipelinebus ausgeben.

Kapitel 4

Field Programmable Gate Arrays FPGA

Die Aufgabe dieser Diplomarbeit war die Realisierung eines so genannten Readout-Mergers durch die Programmierung eines FPGAs.
Dieses Kapitel erläutert die allgemeine technische Natur von FPGAs als auch die Details eines Xilinx Virtex-E FPGA-Chips. Des Weiteren werden die verwendeten Software Werkzeuge vorgestellt und der generelle Designprozess skizziert.

4.1 Technischer Hintergrund

Vor der Einführung von FPGAs gab es zur Lösung von komplexeren Logikaufgaben im Allgemeinen zwei verschiedene Lösungsansätze.
Im Falle von einfacher Logik aber hohen Geschwindigkeitsanforderungen realisierte man die Lösung mit Hilfe von *ASIC-Chips*[19]. Diese Chips werden ausschließlich für einen bestimmten Zweck entwickelt und hergestellt. Sie werden durch Standard-Industrieprozesse auf Siliziumbasis erzeugt. Es handelt sich also um eine *reine Hardwarelösung*.
Bei sehr komplexer Logik und/oder nicht zu hohen Geschwindigkeitsanforderungen empfiehlt sich dagegen die Verwendung von kommerziellen *Mikroprozessoren*. Ein Programmcode ist einfacher und schneller zu entwickeln als das Hardwaredesign eines ASIC. Darüber hinaus kann ein Programm auch beliebig geändert, erweitert oder von Fehlern befreit werden. Es handelt sich also um eine *Softwarelösung*.
Die Verwendung eines *FPGA* stellt dagegen eine Art Vermischung von beiden Welten dar. Ein FPGA lässt sich dynamisch programmieren, sein (programmiertes) Verhalten und seine (hohe) Geschwindigkeit entsprechen allerdings viel mehr dem eines ASIC als dem eines Mikroprozessors.
Dazu gehört unter anderem, dass ein Prozessor einem einzigen sequentiellen Logikstrang folgt (seinem Programm), während ein FPGA oder ASIC beliebig viele Schaltungen parallel ausführen kann.

4.1.1 Grundstruktur eines FPGA

Grundsätzlich besteht ein FPGA aus einer Vielzahl von vier verschiedenen Grundkomponenten:

- **Logikblöcke** bieten die Grundarchitektur für die logische Verarbeitung von beliebigen Signalen (zum Beispiel die Verknüpfung von zwei Eingangssignalen durch ein logisches UND)
- **Ein- und Ausgabeblöcke** ermöglichen die Kommunikation über die zu Verfügung stehenden Pins des FPFA Hardwarechips mit der angeschlossenen externen Elektronik.
- **Speicherblöcke** ermöglichen die Speicherung von Informationen innerhalb des Chips.
- Das **„Routing Network“** ermöglicht die dynamische Verbindung der obigen drei Komponenten und die Verteilung von globalen Takt- und Resetsignalen.

[19] ASIC - Application Specific Integrated Circuit

Diese Komponenten sind innerhalb eines FPGA in einer Art regelmäßigem Gitter angeordnet. Ein FPGA ermöglicht nun die dynamische „Programmierung“ dieser Komponenten und ihre dynamische Verdrahtung durch das Routing Network. Diese Programmierung des FPGA erfolgt im Normalfall ein einziges Mal nach jedem Power-up der entsprechenden Elektronik. Danach ist die Programmierung als statisch anzusehen und kann *während* des Betriebes des FPGA nicht mehr geändert werden.
So können zum Beispiel mehrere Logikblöcke auf bestimmte Weise miteinander verbunden werden. Zusammen mit der Einstellung, welche dieser Blöcke als „UND“, „ODER“ oder „Exklusiv-ODER“ fungieren sollen, lässt sich so beispielsweise die Addierung zweier Integerzahlen realisieren. In der Praxis bietet ein einzelner Logikblock allerdings meist wesentlich mehr Funktionen als lediglich UND / ODER / X-ODER.
Die reine Bestimmung wie Logikblöcke miteinander verknüpft werden müssen um eine gegebene Logikaufgabe zu implementieren ist jedoch nur ein Teilaspekt der korrekten Programmierung eines FPGA.
Ebenso bedeutsam ist die *Platzierung* der verwendeten Komponenten und ihrer Verdrahtung. Es muss darauf geachtet werden, dass die Verbindungen zwischen den einzelnen Komponenten innerhalb des FPGA so kurz wie möglich sind, um so die Laufzeitverzögerungen von Signalen so weit als möglich zu minimieren.

4.1.2 Der Virtex-E von Xilinx Inc.

Der erste kommerzielle FPGA wurde von der Firma Xilinx 1985 auf dem Markt gebracht und bis heute ist Xilinx Marktführer in den meisten Bereichen des rasch wachsenden FPGA-Markts.
In der Nomenklatur von Xilinx wird ein Logikblock *CLB* (Configurable Logic Block) genannt, ein Eingabe/Ausgabeblock wird *IOB* (Input/ Output Block) genannt. Speicherblöcke werden als *Block SelectRAM* bezeichnet. Darüber hinaus verfügt jeder Virtex-E über acht *Delay Locked Loops* (DLL) zur Anpassung von eingehenden Clock Signalen. Zur möglichst synchronen Verteilung von Clock Signalen bietet der Virtex vier globale *Clock-trees*, somit kann intern mit bis zu vier verschiedenen, unabhängigen Taktsignalen gearbeitet werden.

Input/Output Block (IOB)

Die IOBs fungieren als Schnittstelle zwischen den internen CLBs des Virtex und den externen Anschlüssen des FPGA (Pins). Die Virtex IOBs bieten eine große Auswahl an verschiedenen I/O-Standards unter anderem auch den differentiellen Standard LVDS.
Jeder Pin kann als Eingabe, Ausgabe oder zur bidirektionalen Kommunikation verwendet werden. Dabei kann jeder Ausgabepin auch in einen neutralen (hochohmigen) Zustand versetzt werden.
Die maximale Ausgabe liegt bei 24mA für unipolare Signale.

Block SelectRAM

Alle Virtex Modelle verfügen über mehrere interne RAM-Blöcke. Der Aufbau eines solchen RAM-Blocks ist bei jedem Virtex Modell identisch, lediglich die Anzahl der zur Verfügung stehenden Blöcke ist unterschiedlich.
Jedes SelectRAM bietet eine Speicherkapazität für 4096 Bits. Dabei kann das RAM wahlweise mit einer Datenbreite von 1, 2, 4, 8 oder 16 Bits verwendet werden. SelectRAM ist vollständig dual-ported, was bedeutet, dass zwei gleichzeitige Zugriffe auf das RAM möglich sind. Dabei kann der erste Kanal mit einer anderen Taktung und Bitbreite arbeiten als der zweite.
Für den RemFPGA des PPM wird ein Xilinx Virtex-E 1000 verwendet, welcher über 96 SelectRAM Blöcke verfügt, also über 48kB internes RAM.

Delay Locked Loop

Die acht DLLs des Virtex-E ermöglichen neben einen Stabilisierung des eingehenden Clock Signals auch eine optionale Phasenverschiebung um 90, 180 oder 270 Grad als auch eine Verdopplung oder Halbierung der Taktrate. Das Ausgangssignal eines Virtex DLL ist grundsätzlich immer ein gleichmäßiges 50/50 Signal.

4.1.3 Der Entwicklungsprozess

Die Entwicklung eines digitalen Designs für einen FPGA unterteilt sich in mehrere Einzelschritte.

Zu Beginn wird das gewünschte Verhalten des FPGA in einer so genannten *Hardware Description Language* (HDL) programmiert. Im professionellen Einsatz stehen zwei verschiedene Sprachen zur Verfügung – *Verilog* und *VHDL*. Beide Sprachen sind entfernt an die Softwaresprache C angelehnt und bieten grundsätzlich die gleichen Möglichkeiten. Für die Programmierung des RemFPGA wurde Verilog verwendet, da alle anderen Hardwareprojekte der ATLAS-Gruppe in Heidelberg ebenfalls mit dieser Sprache arbeiten.

Nach Abschluss der Programmierung folgt die so genannte *Synthese*. Hierbei wird das HDL Programm kompiliert und in eine *Netlist* umgewandelt. Diese Netlist repräsentiert das HDL Design auf der Ebene von logischen Verknüpfungen (UND, ODER, X-ODER, usw.).

Zum Abschluss wird die Netlist von einem so genannten *Place-and-Route* Werkzeug (PAR) analysiert welches wieder eine (neue) Netlist erzeugt. Die Aufgabe des PAR ist die Zuordnung, Verknüpfung und Positionierung der real im FPGA existierenden CLBs, um die von der Synthese-Netlist geforderte Funktionalität zu implementieren. Diese PAR-Netlist wird dann in einem letzen Schritt in einen Bitstream umgewandelt, welcher direkt in die FPGA Hardware geladen werden kann.

Nach jedem dieser drei Schritte kann das Design mit speziellen Werkzeugen *simuliert* werden, um das ordnungsgemäße Funktionieren sicherstellen zu können. Hierbei simuliert das Werkzeug das komplette FPGA Design, welches in eine so genannte *Testbench* eingebettet wird. Aufgabe der Testbench ist die Generierung von Eingangssignalen für die Simulation, um so die Reaktion und Verarbeitung der eingehenden Signale analysieren zu können. Eine solche Testbench wird ebenfalls in HDL erstellt.

Dabei kommen das gleiche Simulationswerkzeug und die exakt gleiche Testbench für alle drei Testschritte zum Einsatz. Von Schritt zu Schritt erhöht sich lediglich die Detailtreue der Simulation des FPGA Designs.

Wird nur der HDL Code simuliert, so wird lediglich die logische Funktion des Designs simuliert.

Eine Simulation der Netlist nach Place-and-Route dagegen berücksichtigt sämtliche Verzögerungen durch Leitungslängen innerhalb des FPGA und teilweise auch kapazitive Effekte.

Kapitel 5

Der Readout-Merger FPGA (RemFPGA)

Gegenstand dieser Diplomarbeit ist die Entwicklung des Verilog Programmcodes für den RemFPGA des Präprozessormoduls (PPM).
Der RemFPGA ist verantwortlich für sämtliche Konfigurationsprozesse auf dem PPM, wie auch den Empfang und die anschließende Komprimierung von ReadOUT Daten. Für die erforderliche Kommunikation mit der „Außenwelt“ verfügt der RemFPGA über die beiden voneinander unabhängigen Schnittstellen VME-Bus und Pipeline-Bus. Somit ist der RemFPGA von zentraler Bedeutung für die Funktion des Präprozessors. Ohne RemFPGA sind die einzelnen PPMs für die Außenwelt unerreichbar und die Komponenten des PPM nicht konfigurierbar.

5.1 Grundsätzliche Aufgabenstellung

Der RemFPGA hat folgende Aufgaben zu erfüllen:

- **Implementierung von vier SPI Schnittstellen** zur Konfiguration der MAX529 DACs der Analog Input Boards.
- **Implementierung eines I²C-Masters** zur Konfiguration der Phos4 Chips der 16 PPrMCMs.
- **Implementierung von 32 seriellen Schnittstellen** zur Konfiguration und Auslese von ReadOUT Daten von 16 PPrASICs.
- **Komprimierung** der ReadOUT Daten der PPrASICs.
- **Implementierung eines Pipelinebus Slave-Nodes** zur Übertragung der komprimierten ReadOUT Daten.
- **Implementierung einer Schnittstelle zum VME-CPLD** zur Kommunikation über den VME-Bus.

5.1.1 Notwendigkeit der Kompression der Readout Daten

Die PPrASICs des Präprozessors erzeugen bei jedem positiven Level-1 Accept für jeden einzelnen Kanal ReadOUT-Daten. Diese Daten bestehen beim Betrieb mit Standardeinstellungen aus einem ReadOUT-Header Wert, einem BCID Wert und fünf RAW Werten. Jedes PPM verarbeitet 64 Kanäle, somit erzeugt jedes PPM 728 Bytes an ReadOUT Daten pro Level-1 Accept.
Bei einer vollen Level-1 Accept Rate von 100 kHz ergibt sich somit eine Datenrate von etwa 71 MB/s pro PPM.
An den für die Übertragung von ReadOUT-Daten bestimmten Pipeline-Bus sind 8 PPMs angeschlossen, somit beträgt die gesamte Datenrate etwa 568 MB/s.
Der Pipelinebus besitzt eine Datenbreite von 32 Bit, womit sich eine mindestens erforderliche Busfrequenz von 142 MHz ergibt.
Innerhalb des Crates sind die einzelnen PPM und ROD Module über eine nur teilweise abgeschirmte Backplane miteinander verbunden, wobei einzelne Leitungen Längen von bis zu 30 cm erreichen. Bei diesen Randbedingungen ist der Betrieb des Pipelinebusses mit 142 MHz technisch nicht möglich.

Ein Prototyp des Pipeline-Bus wurde erfolgreich mit 40 MHz betrieben [8]. Ein Betrieb mit 60 MHz erscheint ebenfalls möglich wurde jedoch noch nicht getestet. Um somit die generierten ReadOUT Daten übermitteln zu können, ist eine Kompression dieser Daten erforderlich.
Verschiedene Algorithmen zur Reduktionen des Datenumfangs wurden untersucht ([9] und [10]). Das vom RemFPGA letztendlich verwendete Kompressionsverfahren ist von den Ergebnissen dieser Untersuchung inspiriert und wird in Kapitel 6.1 ausführlich erläutert.

5.2 Design des RemFPGA Programmcodes

Dieser Abschnitt erläutert den logischen Aufbau der Programmcodes der RemFPGA
Der Code des RemFPGA besteht aus einer Vielzahl von logischen Blöcken. Einzelne Programmblöcke werden bei Verilog als *Module* bezeichnet. Ein- und Ausgabesignale von Modulen werden *Ports* genannt. Dem Design der einzelnen Module der RemFPGA wurden einige Grundprinzipien zu Grunde gelegt, die auch aus der Welt der konventionellen Softwareprogrammierung bekannt sind (siehe Abb. ??).

(a) Zwischen den einzelnen Modulen bestehen jeweils nur so genannte *horizontale* und *vertikale* Verbindungen und Abhängigkeiten. *Diagonale* Abhängigkeiten sind zu vermeiden.

(b) Mehrere Instanzen eines Modultyps werden nur in ausschließlich einer einzigen horizontalen Ebene verwendet.

(c) Daten fließen in wohl definierte Richtungen, sie ändern auf ihrem Weg durch mehrere Module nicht die Richtung.

(d) Erfolgt ein Datenfluss über mehrere horizontale Ebenen, so werden nach Möglichkeit keine Module übersprungen.

(e) Horizontaler Datenfluss ist erlaubt, allerdings soweit als möglich zu minimieren.

(f) Der interne Zustand eines Moduls ist nur abhängig von direkt verbundenen anderen Modulen.

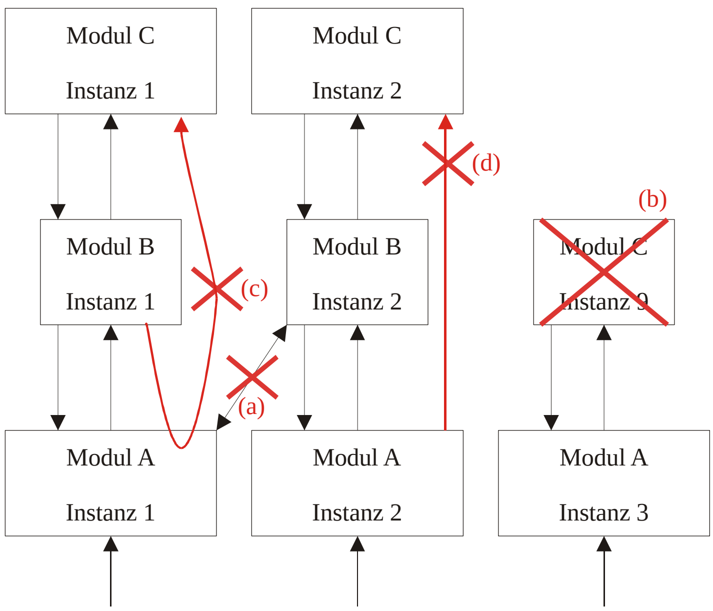


Abbildung 25 - Designregeln

Grundsätzlich enthält jedes Verilog Modul umfassende Kommentare zu der Funktion des Moduls als Ganzem und den erwarteten Eingangs- und Ausgangssignalen. Zusammen mit diesen Designregeln und der folgenden Skizzierung der einzelnen Module ist sichergestellt, dass sich bei Änderungsbedarf am bestehenden Programmcode die Einarbeitungszeit in akzeptablen Grenzen hält.

5.3 Module des RemFPGA

Die folgenden Abschnitte erläutern kurz die Funktion eines jeden Verilog Moduls des RemFPGA und stellen die Beziehungen und Abhängigkeiten zu anderen Modulen dar. Es werden die grundlegenden Ideen und Motivationen für die Wahl der Struktur des jeweiligen Moduls dargelegt, ohne dabei zu sehr ins technische Detail zu gehen. Einige Illustrationen zur Hierarchie dieser Module folgen in Abschnitt 5.4.
Grundsätzlich arbeitet der RemFPGA intern mit einer Taktfrequenz von 64 MHz. Dieses Taktsignal wie auch ein 40 MHz Signal (welches für die seriellen Schnittstellen benötigt wird) werden auf dem PPM erzeugt und von dem VME-CPLD an der RemFPGA geliefert.

5.3.1 RemSerCore

Dieses Modul implementiert die Grundfunktionalität einer seriellen Schnittstelle der PPrASIC.
Dieses Modul implementiert jeweils ein Schieberegister für die Ausgabe und den Empfang von seriellen 13 Bit Worten des PPrASIC. Es arbeitet mit einem 40 MHz Taktsignal welches gleichzeitig auch das Taktsignal der 32 seriellen Schnittstellen darstellt. Ebenfalls erzeugt wird das Frame-Signal, welches dem PPrASIC die Ausgabe bzw. den Empfang eines kompletten 13Bit-Wortes anzeigt.

5.3.2 RemSerSyncDecode

Dieses Modul nimmt eingehende 13Bit-Worte von einem RemSerCore Modul entgegen und synchronisiert diese zur internen RemFPGA Clock von 64 MHz. Jedes RemSerSyncDecode enthält jeweils eine RemSerCore Instanz.
Der Typ eines empfangenen Wortes wird anhand der Flagbits in Bit 12 und 11 und der Reihenfolge im Vergleich zu vorigen Wörtern bestimmt. Auch der Kanal eines ReadOUT Wortes wird bestimmt (zur Erinnerung: jedes serielle Interface überträgt Daten für zwei Trigger-Kanäle). Ein eingegangenes Wort wird eingeordnet als:

- ReadOUT Header
- ReadOUT BCID
- ReadOUT RAW
- ReadBACK Status
- ReadBACK nicht Status

Gleichzeitig wird ein von einem höheren Modul geliefertes Wort über RemSerCore auf der seriellen Schnittstelle ausgegeben und nach vollständiger Übertragung jeweils ein neues Wort vom höheren Modul angefordert.

5.3.3 RemSerInReadout

Dieses Modul verarbeitet die ReadOUT-Daten von einem RemSerSyncDecode Modul, also die Daten von einer seriellen Schnittstelle, welche Informationen für zwei Kanäle überträgt. Dieses Modul komprimiert die eingehenden ReadOUT-Daten und leistet damit eine der Hauptaufgaben des RemFPGA. Die komprimierten Daten werden in einem RAM mit einer Datenbreite von 32 Bit abgelegt.
Der Virtex-E 1000 verfügt über 96 SelectRAM Blöcke mit einer maximalen Datenbreite von 16 Bit. Für 32 Bit sind somit zwei parallel geschaltete SelectRAMs nötig. Der RemFPGA implementiert 32 serielle Schnittstellen die unabhängig parallel arbeiten. Somit würden 32 RemSerInReadout Module insgesamt 64 von 96 SelectRAM-Blöcken benötigen.
Um den RAM-Bedarf zu verringern, teilen sich aus diesem Grund jeweils zwei Module ein 32 Bit RAM. Somit werden insgesamt nur noch 32 Blöcke benötigt. Das RAM befindet sich im Modul RemSerRoCollect (siehe 5.3.5) welches jeweils zwei RemSerRoCompress Module kontrolliert.

Das Modul arbeitet mit einer umfangreichen State-machine[20] deren Zyklus mit dem Empfang des ersten ReadOUT-Headers nach einem ReadBACK Wort beginnt (siehe Abbildung 26).

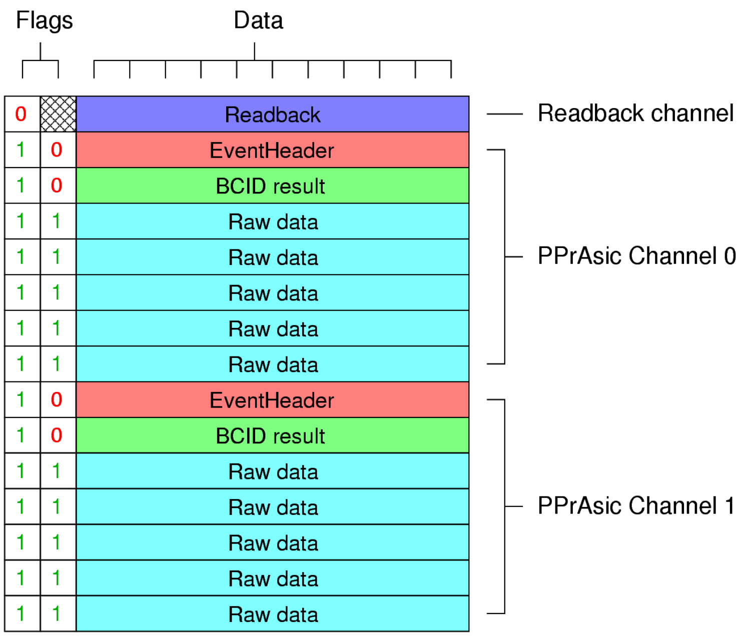


Abbildung 26 - serielles Datenformat

Jeder Zyklus führt zur Speicherung eines Datenpakets von variabler Länge im 32 Bit RAM. Dieses Datenpaket besteht aus einem Header von 18 Bit Länge (siehe Tabelle 4) und einem darauf folgenden Bitstrom von variabler Länge. Die im empfangenen ReadOUT-Header enthaltenen Werte für BCID Nummer und Eventnummer werden im 18Bit-Header abgelegt.

[20] Bei jedem Taktzyklus wird genau eine von mehreren möglichen Aktionen ausgeführt, abhängig von einer Zustandsvariablen (state). Jede Aktion kann einen neuen Zustand für die Ausführung bei dem nächsten Taktzyklus festlegen.

Bit	Bedeutung
0-7	Länge in Bits (inkl. Header)
8-11	BCID-Nummer
12-15	Eventnummer
16	falsche BCID-Nummer
17	falsche Eventnummer

Tabelle 4 - interner 18Bit ReadOUT-Header

Diesen ersten 18 Bits folgt ein Kanal-Header von variabler Länge. Um die gesamte Datenlänge zu minimieren besitzt der Kanal-Header eine variable Länge. An diesen Kanal-Header schließen die komprimierten ReadOUT-Daten für den ersten Kanal des seriellen Interfaces an. Daraufhin folgen die Daten für den zweiten Kanal, ebenfalls bestehend aus einem variablen Kanal-Header und komprimierten ReadOUT-Daten.
Das Modul RemSerInReadout erwartet für jeden Kanal eine bestimmte (über Register konfigurierbare) Anzahl an BCID- und RAW-Werten. Sendet ein PPrASIC eine andere Menge an ReadOUT-Worten, so werden die empfangenen Daten für beide Kanäle der betroffenen seriellen Schnittstelle verworfen und lediglich die Informationen der beiden ReadOUT-Header verarbeitet. Lediglich bei aktiviertem Debug-Modus (siehe unten) werden beliebige Mengen an ReadOUT-Daten akzeptiert.
Die Gesamtlänge des resultierenden Datenpakets darf 256 Bits nicht überschreiten, was jedoch ausreichend für sämtliche möglichen ReadOUT-Konfigurationen des PPrASICs ist. Das genaue Bitschema, das zur Speicherung und Komprimierung verwendet wird, ist in Kapitel 6.1 dargelegt.
Bei ordnungsgemäßer Synchronisation aller Bausteine des Präprozessormoduls werden von allen 32 PPrASICs für jedes Level-1 Accept Signal ReadOUT-Daten mit identischer BCID-Nummer und identischer Eventnummer erzeugt. Somit müssen statt $2 * 32 * 4 = 256$ Bits nur 8 Bits übertragen werden.
Die BCID-Nummer und die Eventnummer des ersten Kanals der seriellen Schnittstelle werden im 18 Bit Header abgelegt und während des direkt folgenden Empfangs der Daten für den zweiten Kanal mit diesen auf Übereinstimmung überprüft. Bei fehlerhafter Synchronisation wird das entsprechende Fehlerbit im Header gesetzt. Diese Informationen werden weiter verwendet von dem Modul RemSerMerge (siehe Abschnitt 5.3.8).

Das Modul RemSerInReadout kann die empfangenen ReadOUT-Daten je nach Konfiguration in drei unterschiedlichen Formaten abspeichern.

Bei *normaler Operation* werden die empfangenen Daten komprimiert und auf Korrektheit überprüft. Die Gültigkeitsüberprüfung ist sehr gründlich und soll eventuelle Datenverfälschungen durch „Cross-talk“ der seriellen Leitungen auf dem Präprozessormodul erkennen helfen. Folgende Fehler werden vom Programmcode erkannt:

- Jeder ReadOUT-Zyklus muss mit einem ReadOUT-Header nach einem ReadBACK-Wert beginnen.
- falsche Reihenfolge von ReadOUT-Header, BCID- und RAW-Werten.
- mehr als zwei ReadOUT-Header zwischen zwei ReadBACK-Werten.
- ReadOUT-Header für 2. Kanal vor dem Header des 1. Kanals.
- Falsche Anzahl von BCID- und/oder RAW-Werten.

Bei *Debug-Operation* werden die eingehenden ReadOUT-Daten weder auf Korrektheit analysiert noch komprimiert. Angefangen mit dem ersten ReadOUT-Wert nach einem ReadBACK-Wert werden die vollen 13 Bit Werte abgespeichert. Der Bitstrom wird durch das den ReadOUT-Daten folgende ReadBACK-Wort abgeschlossen.
Dieser Modus empfiehlt sich vor allem zur Überprüfung der seriellen Kommunikation und zur Erkennung von eventuellen „Cross-talk“ zwischen verschiedenen Leitungen.

Bei *vereinfachter Operation* werden die eingehenden Daten auf Korrektheit überprüft jedoch nicht komprimiert. Die eingehenden BCID- und RAW-Werte werden als umkomprimierte 11 Bit Worte abgespeichert.
Dieser Modus ist besonders nützlich, wenn die ReadOUT-Daten des Präprozessormoduls von Hand analysiert werden sollen oder falls die Software zur Auswertung der Daten in einer noch unvollständigen Form vorliegt.

5.3.4 RemSerInReadback

Dieses Modul verarbeitet die ReadBACK-Daten einer seriellen Schnittstelle.
Die eingehenden Daten werden nicht zu einem Bitstrom komprimiert, da die Menge an anfallenden ReadBACK-Daten während des Triggerbetriebs sehr gering ist. Nur während der Konfigurationsphase vor einem Datenlauf des ATLAS-Detektors fallen größere Mengen an ReadBACK-Daten an. Jedoch ist die Geschwindigkeit der Datenübertragung in dieser Phase nicht von Bedeutung.
Der Verzicht auf Komprimierung vereinfacht den Programmcode erheblich, was zu einem wesentlich geringeren Verbrauch von FPGA-Ressourcen führt. Ein empfangenes Datenwort besitzt eine Breite von maximal 13 Bits. Der Verzicht auf Komprimierung beinhaltet auch das Fehlen eines Bitschieberegisters und somit werden die empfangenen Daten auf 16 Bit Breite aufgefüllt und jeweils in Paaren in einem 32 Bit RAM abgelegt. Dabei teilen sich vier RemSerInReadback Module einen RAM-Block. Somit werden insgesamt 16 Virtex SelectRAMs benötigt (siehe auch Abschnitt 5.3.3).
Der Programmcode erkennt und verarbeitet Blöcke von ReadBACK-Daten bestehend aus einem ReadBACK-Header, gefolgt von einer beliebigen Anzahl von ReadBACK-Daten.
Nicht im RAM gespeichert werden jedoch ReadBACK-Statusworte. Diese werden vom PPrASIC immer dann gesendet, wenn keine anderen ReadBACK-Daten auszugeben sind. Das Modul stellt das jeweils zuletzt empfangene Statuswort für übergeordnete Module zur Verfügung.

5.3.5 RemSerRoCollect

Dieses Modul instanziiert zwei RemSerInReadout Module und stellt diesen Zugriff auf einen 32 Bit RAM-Block zur Verfügung. Weiterhin liefert es übergeordneten Modulen einen Mechanismus zur Auslese dieses Speichers.
Selbst bei deaktivierter Komprimierung beträgt die Größe eines Blocks an ReadOUT-Daten im Normalfall nicht mehr als 6 Speicherworte (bei zwei Kanälen mit je einem Header, einem BCID- und fünf RAW-Werten zu je 13 Bit ergeben sich 182 Bits). Ein SelectRAM des Virtex FPGA besitzt jedoch eine Kapazität von 4096 Bits. Um ein 32 Bit RAM zur Verfügung zu stellen, sind zwei SelectRAMs nötig, somit ergibt sich eine Kapazität von 8192 Bits, welche zur Speicherung von über 40 ReadOUT-Events ausreicht. Daher wird der Speicherblock in zwei gleich große Teile unterteilt und so von zwei RemSerInReadout Instanzen genutzt.
Das SelectRAM des Virtex ist dual-ported, es erlaubt zwei komplett unabhängige gleichzeitige Zugriffe auf seinen Inhalt. Einen Port teilen sich die beiden untergeordneten Module, der zweite Port wird verwendet, um einem übergeordneten Modul die Auslese des Speicherinhaltes zu ermöglichen.

Die komplette Verwaltung dieser Speicherzugriffe wird von diesem Modul realisiert. Die Verwendung von einem geteilten RAM anstelle von zwei unabhängigen Blöcken ist dabei sowohl für die über- wie auch untergeordneten Instanzen nicht ersichtlich und ohne Belang.

5.3.6 RemSerRbCollect

Dieses Modul hat ähnlich wie RemSer*Ro*Collect die Aufgabe einen Speicherbereich für über- und untergeordnete Instanzen zu verwalten. In diesem Fall werden vier SelectRAMs zu einem Speicherblock zusammengefasst.
Vier untergeordnete RemSerInReadback Instanzen teilen sich den Zugriff auf dieses RAM und im Unterschied zu RemSer*Ro*Collect kann jeweils nur ein untergeordnetes Modul Zugriff auf diesen Speicher erlangen. Ein solcher Zugriff besteht aus dem Empfang und der Speicherung eines kompletten ReadBACK-Datenblocks, bestehend aus einem ReadBACK-Header und einer Anzahl von ReadBACK-Daten.
Sollten während dieser Phase auch die anderen drei PPrASICs ReadBACK-Daten senden, so können diese nicht abgespeichert werden und gehen verloren. Ausgenommen sind lediglich Statusworte, welche von RemSerInReadback separat verarbeitet werden (siehe Abschnitt 5.3.4).
Diese Vorgehensweise wird hauptsächlich von der Art der vom PPrASIC gesendeten Daten motiviert. Die Anzahl an ReadOUT-Daten die ein einzelner Auslesebefehl produziert, kann bis zu 8192 Bits betragen (Auslese der PPrASIC-Lookup-Tabelle). Vier SelectRAM-Blöcke bieten 16384 Speicherbits und somit können die Daten von vier seriellen Schnittstellen nicht gleichzeitig abgespeichert werden. Daher ist ein paralleler Zugriff wie bei RemSer*Ro*Collect schon aus Kapazitätsüberlegungen nicht ratsam.
Da ein PPrASIC ReadBACK-Daten jedoch nur als Reaktion auf über die serielle Schnittstelle gesendete Befehle erzeugt kann der Verlust von Daten vermieden werden. Die Steuerungssoftware darf daher nicht alle 32 PPrASICs eines Präprozessormoduls gleichzeitig zur Ausgabe von ReadBACK-Daten auffordern sondern in vier getrennten Schritten jeweils nur jeden vierten PPrASIC.
RemSerRbCollect (8 Instanzen) und RemSer*Ro*Collect (16 Instanzen) verwenden somit insgesamt 64 der 96 verfügbaren Speicherblöcke des Virtex-E 1000.

5.3.7 RemSerOutControl

Dieses Modul nimmt Datenblöcke von einem übergeordneten Modul entgegen und leitet ihren Inhalt an die Ausgabe der 32 seriellen Schnittstellen weiter. Die Datenblöcke werden in einem 32 Bit Speicher abgelegt. Dieser Speicher besteht aus 16 SelectRAM-Blöcken und bietet somit eine Kapazität von 65536 Bits.
Der Inhalt eines Datenpakets kann wahlweise auf einer bestimmten seriellen Schnittstelle ausgegeben werden oder auch auf allen 32 gleichzeitig. Die Möglichkeit der parallelen Ausgabe der gleichen Daten an alle Schnittstellen ermöglicht die perfekte Synchronisation der PPrASICs.

5.3.8 RemSerMerge

Dieses Modul fasst die gesamte Funktionalität der seriellen Schnittstellen zusammen. Es instanziiert 32 RemSerSyncDecode Module, 16 RemSerRoCollect Module, 8 RemSerRbCollect Module und ein RemSerOutControl Modul.

RemSerMerge verbindet die 16 getrennten Datenblöcke der RemSerRoCollect Module zu einem gesamten Bitstrom welcher über den Pipelinebus oder die VME-Schnittstelle ausgelesen werden kann. Bei der Erstellung dieses Gesamtdatenstroms überprüft der Programmcode die erwähnte Übereinstimmung der BCID- und Eventnummern der einzelnen Datenpakete (siehe Abschnitt 5.3.3). Bei fehlerhafter Synchronisation der BCID-Nummer wird dies lediglich mit einem Fehlerbit im Header des resultierenden Gesamtbitstroms angezeigt. Bei fehlerhafter Eventnummer jedoch wird das gesamte Datenpaket verworfen und lediglich der Header mit der Fehlerinformation abgespeichert.

Die weiteren Daten verarbeitenden Instanzen wie der ReadOUT-Treiber (ROD) des Crates oder die auswertende Software verwerten lediglich synchronisierte Eventdaten. Die fehlerhaften Daten werden aus diesem Grund schon innerhalb der RemFPGA verworfen, um so die Anzahl der zu übertragenden und zu speichernden Daten nicht unnötig zu erhöhen.

RemSerMerge verwendet für die Speicherung eines gesamten ReadOUT-Datenpakets einen 32 Bit Speicherblock bestehend aus zwei SelectRAM-Blöcken. Dies bietet ausreichend Platz für die von den einzelnen RemSerInReadout Instanzen erzeugten Datenströme.

RemSerMerge verfügt über zwei dieser Speicherblöcke und kann somit während der Auslese eines Datenblocks durch ein übergeordnetes Modul das nächste Datenpaket im zweiten Speicherblock vorbereiten.

Weiterhin verbindet RemSerMerge die 8 getrennten Datenblöcke der RemSerRbCollect Module. Hierbei kann das resultierende Gesamtpaket jedoch nicht in einem weiteren Speicherblock abgelegt werden, da die mögliche Gesamtlänge dieses Pakets die zur Verfügung stehenden RAM-Ressourcen des Virtex übersteigt. Stattdessen wird das resultierende Gesamtdatenpaket dynamisch während der Auslese aus den Daten der acht einzelnen RemSerRbCollect Module zusammengesetzt. Dies ist möglich, da hier keine Synchronisationsüberprüfungen nötig sind. Auch sind die Längen der acht einzelnen Datenpakete immer ein Vielfaches von 32, was das Aneinanderfügen der Daten wesentlich erleichtert.

Die ReadOUT-Daten wie auch die ReadBACK-Daten werden über einen FIFO-Mechanismus[21] übergeordneten Modulen zur Auslese zur Verfügung gestellt. Über einen weiteren FIFO-Mechanismus leitet RemSerMerge eingehende Daten weiter an die Instanz von RemSerOutControl, um sie so als Ausgabe an die 32 seriellen Schnittstellen weiterzuleiten.

Als Endresultat liefert RemSerMerge Datenleitungen für alle 32 seriellen Schnittstellen, welche direkt mit Ein-/Ausgabe-Pins des FPGA verbunden werden können. Die Aktivität dieser Datenleitungen wird gesteuert durch drei einfache FIFO-Mechanismen und einige Konfigurationsleitungen welche mit internen Registern des RemFPGA verbunden werden.

Die Verbindung der 32 seriellen Schnittstellen mit den 16 RemSerRoCollect und mit den 8 RemSerRbCollect Modulen ist in Abbildung 27 zu erkennen. Gemäß einer Modulodivision werden die beiden Schnittstellen 01/17 mit RoCollect01 verbunden, 02/18 mit RoCollect02 bis zu 16/32 mit RoCollect16. Ebenso wird 01/09/17/25 mit RbCollect01 verbunden bis hin zu 08/16/24/32 mit RbCollect08. Somit können zum Beispiel gleichzeitig ReadBACK-Daten von den Schnittstellen 01 bis 08 oder von auch 01/02/11/20/29/06/07/32 empfangen werden. Nicht jedoch von 01/02/03/04/05/06/07/09, da die Schnittstellen 01 und 09 beide mit RbCollect01 verbunden sind, somit werden die Daten von 09 oder 01 verworfen. ReadOUT-Daten werden immer verarbeitet (siehe Abschnitt 5.3.5 und Abschnitt 5.3.6).

5.3.9 RemI2cControl

Dieses Modul verwaltet den I²C-Bus. Über einen FIFO-Mechanismus nimmt das Modul maximal einen Datenblock entgegen, welcher an angeschlossene I²C-Slaves gesendet wird. Nach der kompletten Ausgabe des Datenblocks kann ein weiterer verarbeitet werden. Der Empfang von Daten über die I²C-Schnittstelle wird nicht unterstützt, da die anzusteuernden Phos4-Chips keine Ausgabedaten erzeugen.

[21] FIFO – „First-In-First-Out", ein Mechanismus zur Pufferung von Daten

RemI2cControl implementiert den Speicherblock bestehend aus zwei SelectRAMs und die Ansteuerung über den FIFO-Mechanismus. Der eigentliche I²C-Master wird implementiert durch kostenlosen und lizenzfreien Programmcode von www.OpenCores.org [11]. Dieses Internetportal bietet eine ganze Reihe von kostenlosen Verilog- und VHDL-Modulen zur freien Verwendung. Der verwendete I²C-Master Quellcode[22] steht seit zwei Jahren zur Verfügung und ist ausführlich getestet worden. Er wird von RemI2cControl unverändert übernommen.

5.3.10 RemSpiCore

Dieses Modul implementiert einen SPI-Bus. Der SPI-Bus kann lediglich Daten senden aber nicht empfangen. Das Modul sendet daher ein 16 Bit Wort auf den Bus und fordert nach Abschluss des Zyklus von einem übergeordneten Modul ein neues Ausgabewort an.
Der SPI-Bus wird mit einer Geschwindigkeit von 1 MBit/s betrieben.

5.3.11 RemSpiControl

Dieses Modul verwaltet vier SPI-Schnittstellen, implementiert durch vier RemSpiCore Instanzen. Ein mit RemI2cControl identischer FIFO-Mechanismus wird zum Empfang maximal eines Datenblocks verwendet.

5.3.12 RemRegControl

Dieses Modul implementiert und verwaltet sämtliche internen Register des RemFPGA. Alle Register besitzen eine Breite von 32 Bit und können über ein Adressschema angesteuert werden. Um die Register sowohl über den VME-Bus wie auch den Pipelinebus lesen und schreiben zu können, bietet RemRegControl zwei identische Zugriffsschnittstellen.

5.3.13 RemVmeCore

Dieses Modul implementiert die „Easy VME“-Schnittstelle (siehe 3.3.4) und synchronisiert den Datenverkehr mit der internen Clock des RemFPGA. RemVmeCore definiert einen Adressraum für den direkten Zugriff auf die internen Register des RemFPGA und jeweils einen FIFO-Mechanismus zur Eingabe und Ausgabe von Daten.
Der Ausgabe-FIFO dient der Auslese von ReadOUT- und ReadBACK-Daten- der Eingabe-FIFO nimmt Daten für die I²C-Bus, für die SPI-Schnittstellen und für die Ausgabe über die seriellen Schnittstellen entgegen. RemVmeCore steuert lediglich den Datentransfer, der Inhalt der Daten und der Weitertransport innerhalb des RemFPGA wird von dem übergeordneten Modul RemMainControl gesteuert.
Das „Easy VME“-Protokoll definiert zusätzlich eine Interrupt-Leitung. Diese wird nicht von RemVmeCore kontrolliert sondern von RemRegControl über den Inhalt von verschiedenen Registern gesteuert.

5.3.14 RemPipeCore

Dieses Modul implementiert den Pipelinebus. Dieser Bus arbeitet mit einem vom ROD erzeugten Taktsignal, dessen Frequenz ähnlich der des RemFPGA ist. Eine der Hauptaufgaben dieses Moduls ist daher eine sichere Synchronisation selbst bei einer sich ständig ändernden Phasenverschiebung zwischen den beiden Taktsignalen.

[22] I²C-Code verwaltet von Richard Herveille - http://www.opencores.org/people/rherveille

RemPipeCore verarbeitet alle eingehenden Pipelinebus-Befehle. Es empfängt oder sendet Daten mit Hilfe von BeginOfData/EndOfData Befehlen und erkennt und verarbeitet Parityfehler. Ähnlich wie RemVmeCore transportiert dieses Modul lediglich Datenpakete, sowohl der Inhalt als auch der Zielort wird vom übergeordneten Modul RemMainControl bestimmt.

5.3.15 RemMainControl

Dieses Modul kontrolliert die VME-Schnittstelle und den Pipelinebus. Hierzu wird jeweils eine Instanz von RemVmeCore und RemPipeCore erzeugt. Es verbindet die Teile des RemFPGA, die lokale Schnittstellen auf dem Präprozessormodul kontrollieren (seriell, I²C und SPI), mit den Schnittstellen, die der Kommunikation mit anderen Einschüben des Präprozessor-Crates dienen (Pipelinebus mit ROD-Einschub und VME mit Crate-Controller).
Somit bildet dieses Modul das Zentrum des gesamten Programmcodes, auch wenn die Hauptaufgaben des RemFPGA wie zum Beispiel Kompression von anderen Modulen erfüllt werden.

5.3.16 RemMain

Dieses Modul fasst den gesamten Programmcode des RemFPGA zusammen. Es implementiert selbst keinerlei eigene Funktionalität, sondern verbindet die Datenleitungen der fünf Schlüsselmodule *RemSerMerge*, *RemI2cControl*, *RemSpiControl*, *RemRegControl* und *RemMainControl* und stellt somit den Funktionsumfang des gesamten RemFPGA zur Verfügung.
Die Portdefinitionen von RemMain sind die Eingangs- und Ausgangssignale des RemFPGA und müssen lediglich noch mit Pins des FPGA-Chips verbunden werden.

5.3.17 RemTop

Dieses Modul verbindet die Datenleitungen von RemMain mit Xilinx Virtex IOB-Komponenten und konfiguriert die DLLs des Xilinx für die Verarbeitung und Chip-weite Verteilung der verschiedenen Clocksignale. Es stellt das so genannte „Top-Level“-Modul dar, welches alle anderen Module des Programmcodes enthält. Innerhalb dieses Moduls können zum Beispiel wahlweise differentielle LVDS Eingabe/Ausgabeblöcke instanziiert werden oder normale unipolare IOBs. Auch die genau Pinbelegung kann hier spezifiziert werden. Diese Einstellungen haben jedoch keinen weiteren Effekt auf die anderen Module des RemFPGA und können beliebig geändert werden.

5.4 Schematische Illustration des Designs des RemFPGA

Abbildung 27 zeigt ein Schema des gesamten RemFPGA. In dieser Zeichnung sind alle Instanzen der verschiedenen Module enthalten. Es ist weiterhin dargestellt, welche Module welche Instanzen anderer Module enthalten. Ebenfalls dargestellt sind die wichtigsten Modulverbindungen und die Richtung des Datenflusses in ihnen. Nicht enthalten sind die Virtex-spezifischen Elemente innerhalb des Topmoduls RemTop.

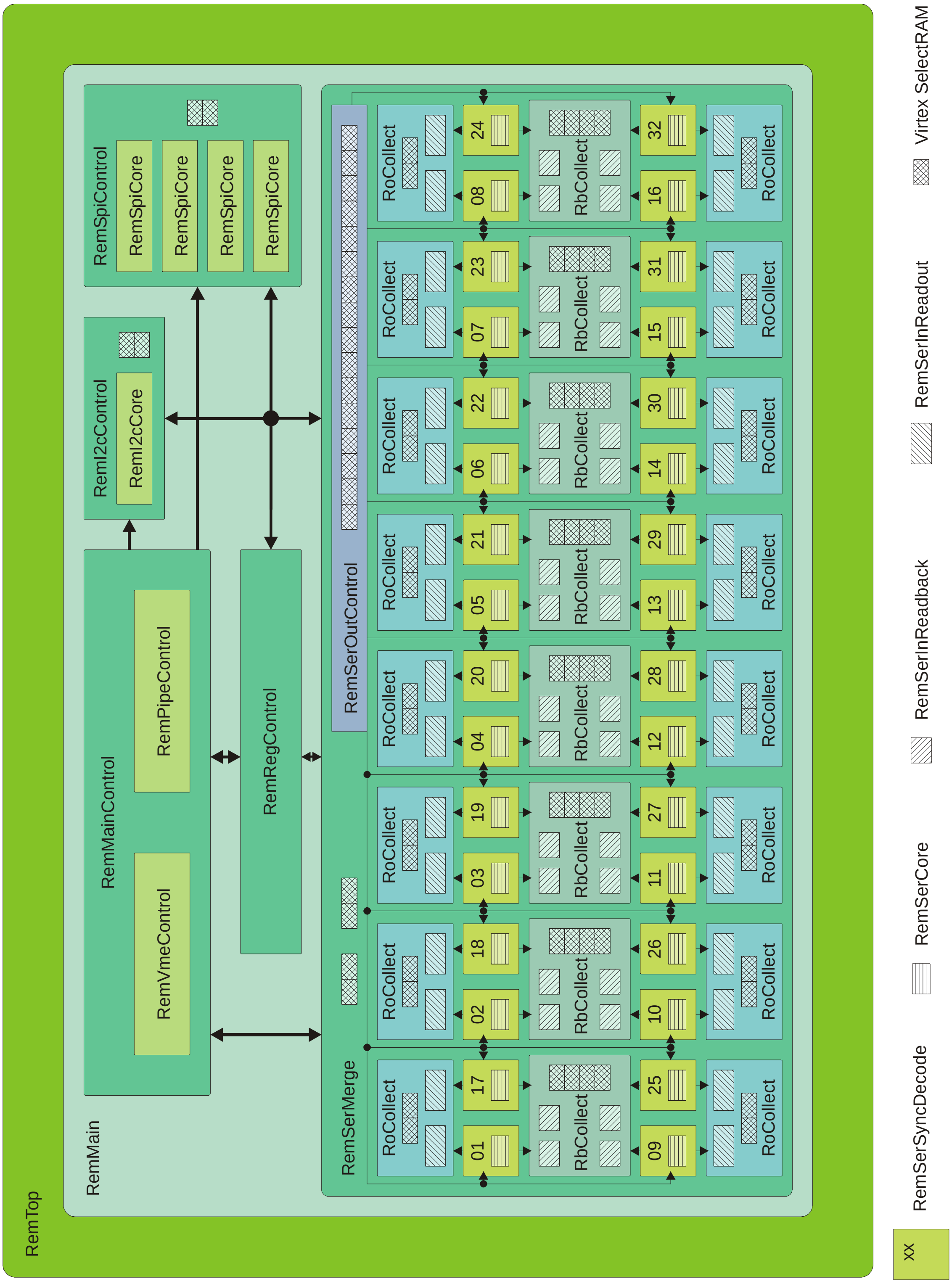


Abbildung 27 - Schema des gesamten RemFPGA

Kapitel 6

Datenformate und Register des RemFPGA

Dieses Kapitel erläutert die verschiedenen Formate der Daten, welche über die VME-Schnittstelle oder den Pipelinebus empfangen oder ausgegeben werden können. Auch die Funktionen und die Inhalte der verschiedenen internen Register des RemFPGA werden dargelegt.

6.1 ReadOUT-Daten und ihre Kompression

In diesem Abschnitt werden zunächst kurz die Ergebnisse der Untersuchungen zu verschiedenen Kompressionsverfahren für ReadOUT-Daten von Bernhard Niemann [9] und Volker Schatz [10] erörtert. Im Anschluss wird die von RemFPGA eingesetzte Kompression ausführlich erläutert und das daraus resultierende Datenformat erklärt.
Der RemFPGA kann mehrere komplette ReadOUT-Datenblöcke in seinem internen Speicher ablegen (je nach Kompression 15 bis 30 Stück). Jede Aufforderung an den RemFPGA ReadOUT-Daten auszulesen, führt jedoch nur zur Übertragung eines solchen Datenblocks. Bei korrekter Synchronisation der PPrASICs liefert somit ein Auslesezyklus die Daten eines Level-1 Accept Ereignisses.

6.1.1 Untersuchte Kompressionsverfahren

Bernhard Niemann entwickelt und untersucht in seiner Diplomarbeit vier verschiedene Algorithmen ausschließlich zur Kompression von ReadOUT-RAW Werten:

Huffmann Kodierung

Dieses Verfahren bildet eine der Grundlagen der auf Personal Computern weit verbreiteten „zip" oder „gz" Kompression. Über einen Binärbaum wird jedem möglichen Wert ein binärer Code von variabler Länge zugewiesen. Häufig auftretende Werte erhalten einen kurzen Binärcode, seltene Werte einen längeren Code.
Dieses Verfahren ist bei einem optimal gewählten Binärbaum äußerst leistungsstark. Dieser Binärbaum stellt jedoch im Fall des RemFPGA ein Problem dar. Ein FPGA verfügt nicht über ausreichende Leistung, um einen korrekten Binärbaum selbst dynamisch errechnen zu können. Dies bedeutet, dass der Baum über einen Konfigurationsschritt dem RemFPGA mitgeteilt werden muss.
Ein Binärbaum basiert auf der statistischen Analyse der zu erwartenden Werte. Sollten sich die Bedingungen des Detektors und damit die Statistik der Werte auch nur leicht ändern so bricht das gesamte Huffmann Verfahren zusammen da der Algorithmus sehr empfindlich gegenüber Änderungen des Binärbaumes ist.

Längenkodierung

Dieses Verfahren ersetzt aufeinander folgende identische Werte durch die Angabe der Länge der Folge und ihren Wert. Tausend Nullen werden somit durch nur zwei Werte (1000 / 0) repräsentiert.

Nur sehr wenige Einzelkanäle werden bei jedem „Bunch-Crossing“ durch auftreffende Teilchen ausgelöst. Somit eignet die Längenkodierung hervorragend für die Kompression der Kanäle welche lediglich einen Nullwert gemessen haben. Allerdings funktioniert dieses Verfahren ausschließlich in der Abwesenheit von Nullpunktrauschen.

Huffmann inspirierte Kodierung (absolutes Verfahren)

Dieses Verfahren ist speziell auf die Struktur der RAW-Daten zugeschnitten und orientiert sich an der Idee Huffmanns, häufige Werte durch kurze Codes zu repräsentieren.
Besonders häufig treten niedrige Energiewerte auf (vor allem Nullpunktrauschen) und die leeren oberen Bits dieser Werte können ausgelassen werden. Der so entstandene Algorithmus verwendet zwei verschiedene Codetypen – ein Codewort voller Länge für hohe Energiewerte und ein Codewort kurzer Länge für niedrige Energiewerte. Die Unterscheidung zwischen den beiden Typen kann zum Beispiel durch ein vorangestelltes Flagbit erfolgen.
Dieses Verfahren benötigt keine externe Konfiguration und ist auch gegenüber Rauschen unempfindlich, solange dieses nicht stärker als der Wertebereich des kurzen Codes ist. Das Verfahren ist allerdings anfällig gegenüber einer Nullpunktsverschiebung der Daten. Je größer die Verschiebung, desto wirkungsloser wird die Verwendung des kurzen Codes, da ein immer größerer Teil des kurzen Wertebereichs nicht mehr genutzt wird.

Huffmann inspirierte Kodierung (Differenzverfahren)

Dieses Verfahren ist eine angepasste Form von vorigem Algorithmus. Um auch robust gegenüber von Nullpunktsverschiebungen zu werden, wird jeweils die Differenz zum vorangegangenen Wert gebildet und in einem kurzen oder langen Code abgelegt. Allerdings ist dieses Verfahren anfälliger gegenüber von Nullpunktrauschen (halbierter Wertebereich).

Verbesserte Huffmann inspirierte Kodierung (beide Verfahren)

Volker Schatz befasst sich in seiner Diplomarbeit [10] unter anderem mit der Analyse der obigen Komprimierungsverfahren und schlägt als Verbesserung unter anderem die Verwendung von drei Codes (kurz / mittel / voll) vor. Hinzu kommen noch einige Details zur möglichst sparsamen Identifizierung dieser drei Typen durch Flagbits.

Fazit der verschiedenen Verfahren

Die ersten beiden Kompressionsverfahren können aufgrund ihrer starken Instabilität gegenüber äußeren Einflüssen nicht sinnvoll zur Kompression eingesetzt werden. Die beste Leistung wird nach V. Schatz [10] durch das verbesserte Huffmann Verfahren erzielt (siehe Tabelle 5):

absolutes Verfahren			
Nullstelle → / Rauschen (σ) ↓	0	5	10
2	2,84	2,52	1,74
4	2,80	2,43	1,86
6	2,74	2,39	1,94

Differenzverfahren			
Nullstelle → / Rauschen (σ) ↓	0	5	10
2	2,62	2,34	2,22
4	2,52	2,15	2,01
6	2,41	2,05	1,88

Tabelle 5 - Kompressionsraten des verbesserten Verfahrens nach [10]

Neuere Untersuchungen der zu erwartenden Energiewerte und die Möglichkeit einer Nullstellenkorrektur durch die Analog Input Module (siehe 3.2.1) stellen sicher, dass die Nullpunktverschiebung der Messwerte garantiert kleiner als 5, wahrscheinlich sogar kleiner als 2 sein wird. Das Rauschen wird ebenfalls kleiner als 5 sein. Somit ergibt sich als bestes Kompressionsverfahren für die RAW-Werte der ReadOUT-Daten die *verbesserte Huffmann inspirierte Kodierung* im *absoluten Verfahren*.

6.1.2 Kompressionsverfahren des RemFPGA

Kompression von RAW-Werten

Das von Volker Schatz entwickelte Verfahren wird unverändert übernommen. Es wird eine per Registerwert definierte Anzahl von aufeinander folgenden RAW-Werten komprimiert, in den folgenden Beispielen werden fünf RAW-Werte verwendet.

Der Algorithmus verwendet drei verschiedene Codelängen. Der lange und der mittlere Typ werden durch zwei jedem dieser Codewörter vorangestellte Flagbits identifiziert. Im Gegensatz dazu wird eine Folge von aufeinander folgenden kurzen Codes nur durch ein einzelnes, der gesamten Folge vorangestelltes Flagbit gekennzeichnet. Ein spezieller reservierter Bitwert der kurzen Codewörter fungiert als Endcode und kennzeichnet das Ende eine solcher Folge (siehe Abbildung 28). Ist jedoch der letzte RAW-Wert als ein kurzer Code abgelegt, so entfällt dieser Endcode.

Zusammen mit jedem 10 Bit RAW-Wert sendet der PPrASIC auch das so genannte „External BCID“ Flagbit, welches nicht komprimiert werden kann und welches daher zu jedem der drei Codetypen als zusätzliches Bit hinzugefügt werden muss. In der Illustration ist dieses Bit jeweils mit „E“ bezeichnet. Lediglich der Endcode 111 eines kurzen Codes verfügt über kein weiteres E-Bit. Abbildung 28 illustriert das Verfahren mit zwei Beispielen.

In *Beispiel 1* werden 5 RAW-Worte mit den Werten 0 / 2 / 3 / 6 / 1 komprimiert. Alle Werte werden als kurze Codes abgelegt und da das letzte Wort kurz ist, ist kein Endcode erforderlich. Somit werden 55 originale Bits auf nur 21 Bits komprimiert, was einer Kompressionsrate von 2,6 entspricht.

In *Beispiel 2* werden die Werte 2 / 6 / 7 / 365 / 6 kodiert. Die ersten beiden Werte werden kurz kodiert, mit einem darauf folgenden Endcode. Der dritte Wert wird als mittlerer Typ abgelegt, da der Wert 7 bei kurzen Codes als Endzeichen reserviert ist. Der vierte Wert wird als voller Code gespeichert und der letzte Wert wiederum als kurzer Code. Auch hier folgt am Ende kein Endcode für den kurzen letzten Wert 6. In diesem Fall werden die eingehenden 55 Bits auf 38 Bits komprimiert, was einer Kompressionsrate von 1,4 entspricht.

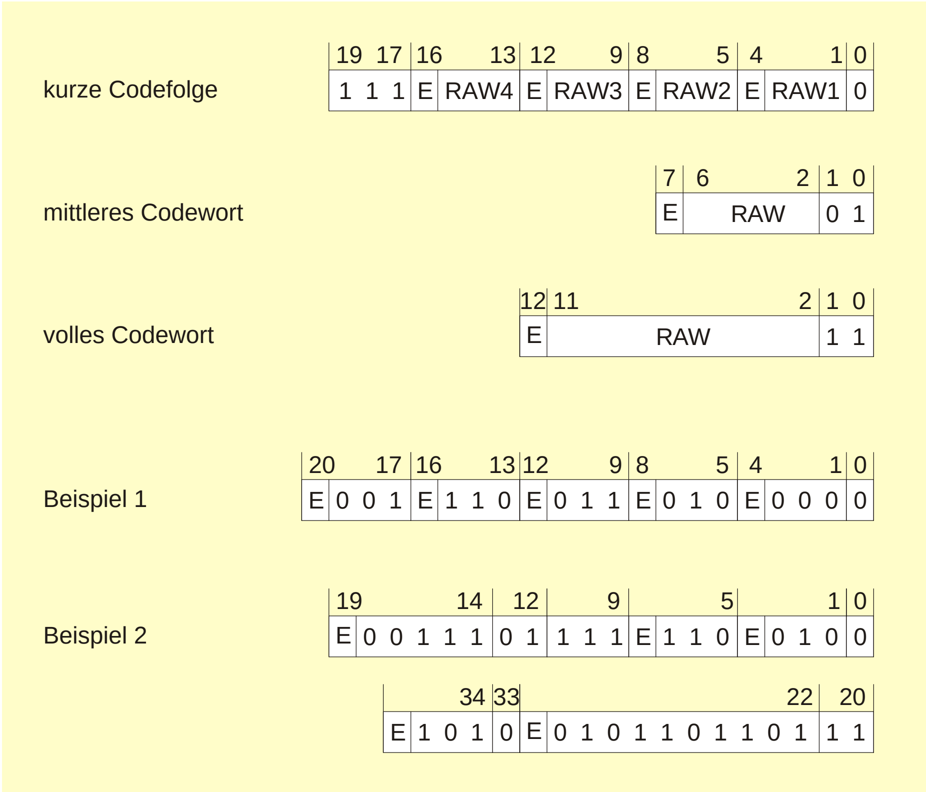


Abbildung 28 - ReadOUT-RAW Kompression

Kompression von BCID-Werten

BCID-Werte wurden in keiner der erwähnten Arbeiten analysiert, da es keine statistischen Werte für ihre Struktur gab bzw. gibt. Trotzdem bietet sich ebenfalls eine Huffmann inspirierte Kodierung an, denn auch hier besteht der Großteil aus Nullwerten.
BCID-Werte sind lediglich 8 Bit breit, daher wurden lediglich zwei Codetypen verschiedener Länge gewählt. Der erhöhte Bitbedarf zur Unterscheidung dreier Codetypen führt bei einer maximalen Codelänge von 8 Bits zu keiner Verbesserung der Kompression. Die Unterscheidung der beiden Typen kann daher durch nur ein Flagbit erfolgen, welches jedem Codewort vorangestellt wird. Eine Verwendung von Endcodes wie bei kurzen RAW-Codes ist nicht erforderlich. BCID-Werte sind 8 Bits breit, wozu jedoch pro Wert noch drei Flagbits kommen, welche nicht komprimiert werden können (hier bezeichnet als A, B und C). Abbildung 29 illustriert die beiden Codetypen. BCID-Werte sind nicht mit Rauschen behaftet, daher wurde der kurze Typ mit einem sehr kleinen Wertebereich versehen, um die Nullmessungen der Detektorkanäle optimal komprimieren zu können.

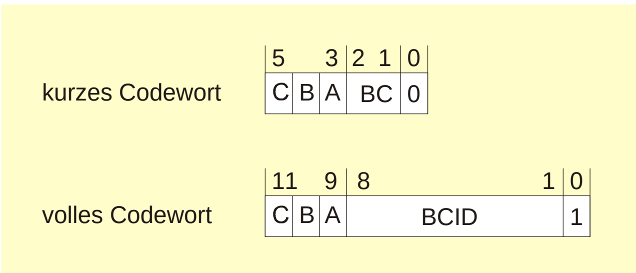


Abbildung 29 - ReadOUT-BCID Kompression

Bei Betrieb des PPrASIC mit Standardparametern wird ein BCID-Wert pro ReadOUT-Datensatz generiert. Damit ergibt sich ein Kompressionsfaktor von 1,8 oder 0,9 je nach verwendetem Codetyp.

Kompression des ReadOUT-Headers und des Kanal-Headers

Abbildung 30 zeigt das Format des vom PPrASIC für jeden Kanal gesendeten ReadOUT-Headers. Die kombinierten Headerdaten aller 64 Kanäle eines Präprozessormoduls enthalten viele redundante Daten. Wie in Kapitel 3.3.3 und Kapitel 5.3.3 erläutert, müssen die Werte der 64 Eventnummern und der 64 BC-Nummern übereinstimmen. Das Kanalbit (Channel) ist für die weitere Verarbeitung unwichtig, da der RemFPGA alle 64 Kanäle in einer definierten Reihenfolge kodiert. Das „Lossflag“ ist durch einen Designfehler des PPrASIC immer null und kann daher ignoriert werden. Somit verbleibt nur das „Headers Only“-Flag.

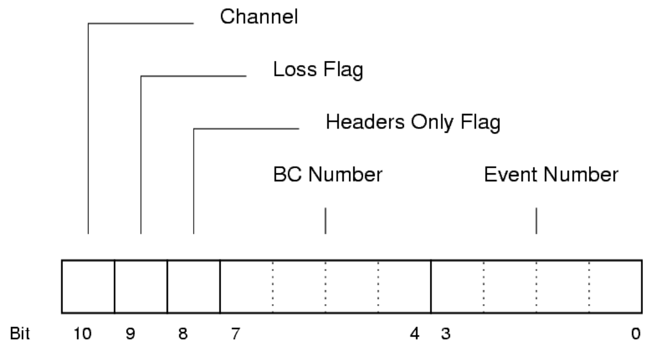


Abbildung 30 - Format der ReadOUT-Headers

Die Verarbeitung der Daten durch den RemFPGA erfordert allerdings die Verwendung einiger zusätzlicher Flagbits pro Kanal. Bei all diesen Bits handelt es sich um Fehlerflags oder Bits für besondere Betriebsmodi. Im normalen fehlerfreien Betrieb sind alle Bits nicht gesetzt. Daher wird vom RemFPGA für jeden Kanal ein Header variabler Länge erzeugt. Im fehlerfreien Fall ist lediglich ein „alles in Ordnung“ Bit nötig (siehe Abbildung 31 und Kapitel 5.3.3).

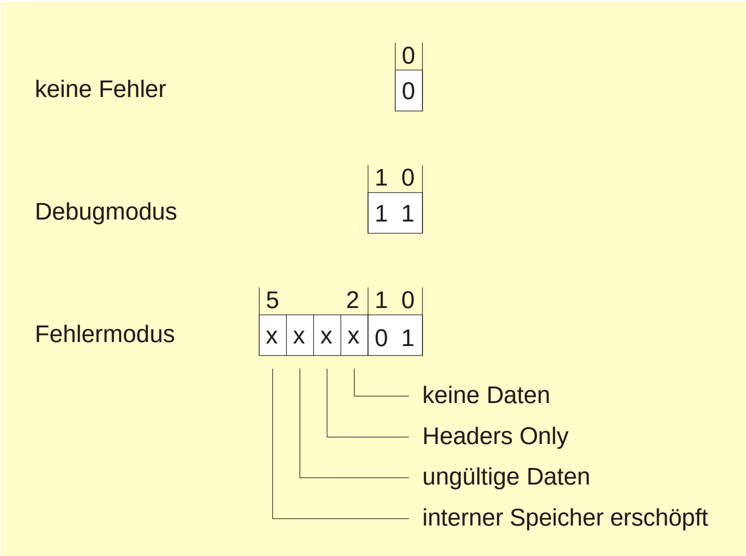


Abbildung 31 - Header eines ReadOUT-Kanals

Ist der Debugmodus aktiviert, so werden alle empfangenen ReadOUT-Daten unverändert und unkomprimiert abgespeichert. Da somit die verarbeitenden Schritte innerhalb des RemFPGA deaktiviert sind, sind auch keine weiteren Flagbits nötig. Bei deaktiviertem Debugmodus kommen bei Fehlern während der Datenverarbeitung vier mögliche Flagbits zum Einsatz.

- **Keine Daten** wird gesetzt falls von einem PPrASIC für einen Kanal überhaupt keine Daten gesendet wurden.
- **Headers Only** erfüllt zusammen mit *Speicher erschöpft* zwei Funktionen. Ist Speicherplatz vorhanden, so ist dieses Bit identisch mit dem *Headers Only* Flag des ReadOUT-Header. Es wurde für diesen Kanal nur ein Header und keine RAW- oder BCID-Werte empfangen. Bei fast vollständig belegtem Speicher verwirft der RemFPGA alle Daten außer dem ReadOUT-Header.
- **Ungültige Daten** wird gesetzt wenn der RemFPGA Daten in einer ungültigen Anzahl oder Reihenfolge empfängt (siehe auch Kapitel 5.3.3).
- **Speicher erschöpft** wird gesetzt, wenn der interne Eventspeicher des Moduls RemSerRoCollect zur Neige geht. (siehe Kapitel 5.3.3 und 5.3.5).

6.1.3 Resultierendes ReadOUT Datenformat

Die Komprimierung der ReadOUT-Daten erfolgt innerhalb des Moduls RemSerInReadout in Verbindung mit dem Modul RemSerMerge (siehe 5.3). Bei der Auslese eines ReadOUT-Datenpakets über die VME-Schnittstelle oder den Pipelinebus verbindet der RemFPGA 32 von jeweils einer Instanz von RemSerInReadout erzeugten Datenpakete zu einen gesamten Bitstrom und versieht diesen mit einem vorangestellten Header von 18 Bit Länge (siehe Abbildung 32).

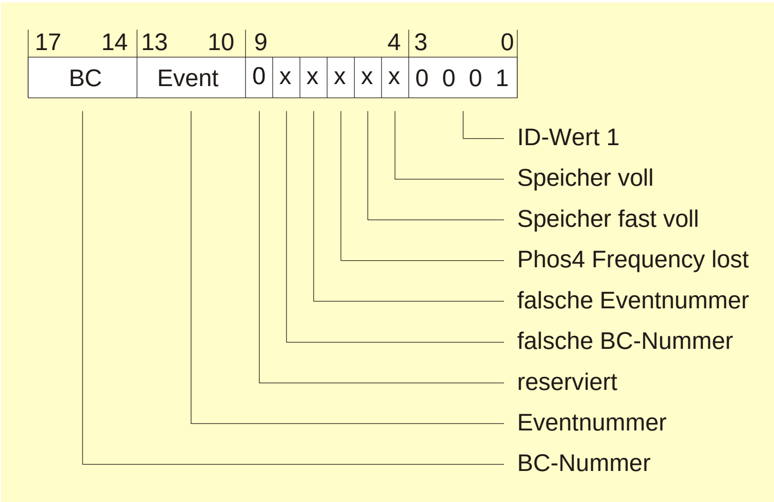


Abbildung 32 - Header der gesamten ReadOUT-Daten

Jeder Datenstrom, der über VME oder Pipelinebus transportiert wird, enthält in den ersten vier Bits einen ID-Code, welcher die folgenden Daten typisiert. Einem RemFPGA ReadOUT-Datenblock ist der ID-Wert 1 zugeordnet. Die acht höchstwertigen Bits enthalten die Event- und „Bunch-Crossing“-Nummer des ersten Kanals der ersten seriellen Schnittstelle. Die Werte der anderen Kanäle werden, wie im vorigen Abschnitt erläutert nicht übertragen da sie bei korrekter Synchronisation identisch sein müssen. Sollte eine Übereinstimmung nicht gegeben sein, so wird das entsprechende *falsche Nummer* Fehlerbit im Header gesetzt. Sollten die Eventnummern nicht korrekt sein, so verwirft der RemFPGA alle ReadOUT-Daten und überträgt lediglich den 18 Bit Header.

Die *Speicher*-Fehlerbits werden gesetzt, wenn Daten von mindestens einer seriellen Schnittstelle nicht abgespeichert werden konnten, da der entsprechende interne Speicherblock des RemFPGA voll oder fast voll war (siehe 5.3.3 und 5.3.5). Diese beiden Bits beziehen sich nicht auf den Zustand des Speichers zurzeit des Empfangs der ReadOUT-Daten die dieser Header beschreibt. Stattdessen werden diese beiden Bits erst zum Zeitpunkt der Auslese über den VME-Bus oder den Pipelinebus gesetzt. Sie informieren daher über den momentanen Zustand des RemFPGA.
Bei einer völlig erschöpften Speicherkapazität werden neue ReadOUT-Daten komplett verworfen. In diesem Zustand ist es für auslesende Instanzen nicht ersichtlich ob, und wie viele ReadOUT-Datensätze verworfen wurden. Bei einer lediglich nahezu erschöpften Speicherkapazität speichert der RemFPGA lediglich die Informationen aus ReadOUT-Headern, BCID- und RAW-Werte werden verworfen. Das *Phos4*-Fehlerbit gibt Auskunft, ob mindestens einer der 32 Phos4-Chips die Synchronisation mit den LHC Taktsignal verloren hat. Diese Fehlerinformation erhält der RemFPGA als Teil der ReadBACK-Statuswörter. Es wird hier den ReadOUT-Daten hinzugefügt, da davon auszugehen ist, dass bei verlorener Synchronisation der PPrASIC fehlerhafte Daten erzeugt.
Abbildung 33 zeigt die ersten drei 32 Bit Worte, wie sie bei einer Auslese über den Pipelinebus oder auch den VME-Bus entstehen. Es sind die empfangenen und komprimierten Daten der beiden Kanäle der ersten seriellen Schnittstelle abgebildet. Es wurden jeweils ein BCID-Wert und fünf RAW-Werte empfangen. Die Eventnummer aller 64 Kanäle beträgt 8 ('b0100), die BC-Nummer ist ebenfalls 8. Der variable Kanal-Header besitzt in den beiden abgebildeten Fällen nur die Minimallänge 1 was durch den Bitwert 0 angezeigt wird. Darauf folgt der BCID-Wert des ersten Kanals mit einem Wert von 33 ('b00100001) und gesetztem Flagbit B. Daraufhin folgen die fünf RAW-Werte 3 / 6 / 7 / 256 / 5. Bei dem letzten RAW-Wert ist das E-Flag gesetzt. Der Kanal-Header des zweiten Kanals besitzt ebenfalls nur die Länge 1. Daraufhin folgen in analoger Weise die Auslesedaten der restlichen 63 Kanäle.

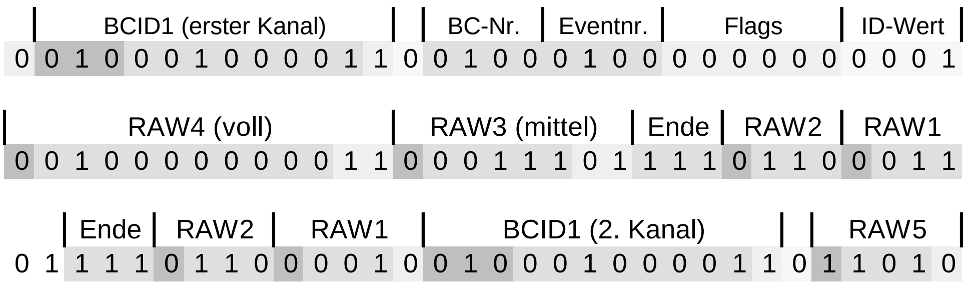


Abbildung 33 - Beispiel eines ReadOUT-Datenstroms

6.2 ReadBACK-Daten

Dieser Abschnitt beschreibt die Datenformate zur Übertragung von ReadBACK-Daten. Diese Datenauslese ist nicht zeitkritisch, weshalb auf Kompression verzichtet wird.
Anders als bei ReadOUT-Daten wird bei einer Übertragung der gesamte Inhalt des RemFPGA-internen ReadBACK-Speichers ausgelesen. Der gesamte ReadBACK-Datenblock besteht somit aus mehreren Unterblöcken. Die Abschnitte 6.2.1 und 6.2.2 beschreiben das Format dieser Untereinheiten, Abschnitt 6.2.3 beschreibt das Gesamtformat.

6.2.1 Format der Lookup-Tabelle des PPrASIC

Der PPrASIC besitzt für jeden Kanal eine interne Lookup-Tabelle, um den eingehenden 10 Bit Rohdaten einen kalibrierten 8 Bit Wert zuzuweisen (unter anderem Nullstellenkorrektur und Rauschunterdrückung, siehe [12]). Diese Tabelle besteht folglich aus 1024 8 Bit Werten. Für die Auslese dieser Tabelle verwendet der RemFPGA ein spezielles Format.
Der Beginn eines Unterblocks beginnt mit einem Headerwort, das eine Breite von 32 Bit besitzt. Diesem Header schließen sich die empfangenen 8 Bit Werte der Lookup-Tabelle direkt an (siehe Abbildung 34 und Abbildung 36).

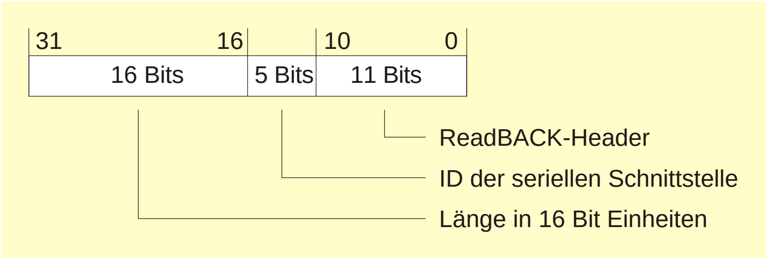


Abbildung 34 – Header von ReadBACK Unterblöcken

Die unteren 11 Bits enthalten den originalen ReadBACK-Header wie er vom PPrASIC über die serielle Schnittstelle gesendet wurde. Er beschreibt den Typ der folgenden ReadBACK-Daten eindeutig (siehe Kapitel 3.3.3). Die nächsten 5 Bits enthalten die Nummer der seriellen Schnittstelle welche diesen Unterblock geliefert hat. Die Länge des Unterblocks einschließlich des Headers wird in 16 Bit Einheiten in den verbleibenden Wertebereich eingetragen.

6.2.2 Alle anderen ReadBACK-Daten

Alle anderen ReadBACK-Daten werden vom RemFPGA in 16 Bit Worten abgespeichert. Die Breite der eigentlichen Daten beträgt lediglich 11 Bit, die restlichen 5 Bits werden mit Nullen gefüllt. Diesen Daten geht ebenfalls der in 6.2.1 beschriebene Header voran. Der darin enthaltene ReadBACK-Header erlaubt die Unterscheidung zwischen Lookup-Tabelle und anderen Daten.

6.2.3 Resultierendes ReadBACK Datenformat

Den einzelnen Unterblöcken geht ein Gesamtheader voraus, welcher eine ReadOUT-ähnliche Struktur besitzt (siehe Abbildung 35). Diesem Header schließen sich die einzelnen Unterblöcke mit ihren eigenen Headern an. Dabei wird mit einer Datenbreite von 32 Bit gearbeitet. Ein neuer Unterblock beginnt immer an einer 32 Bit Adresse, der Zwischenraum wird mit Nullen aufgefüllt (siehe Abbildung 36).

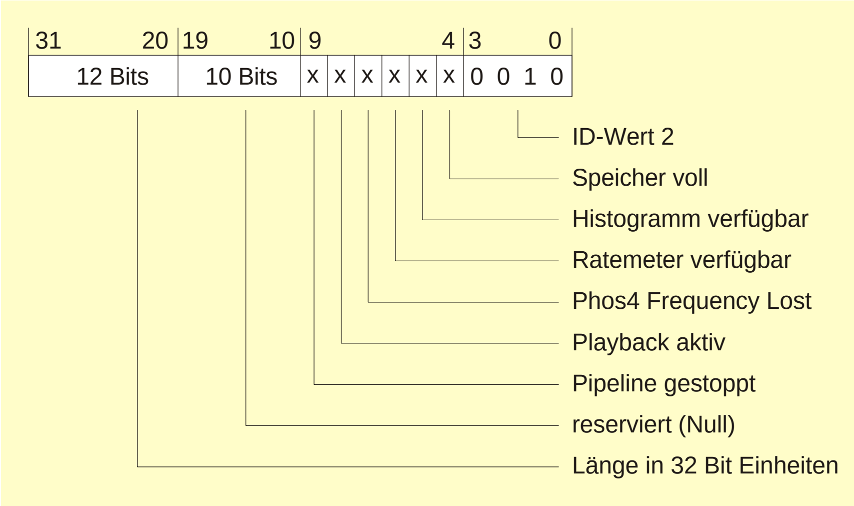


Abbildung 35 - Header der gesamten ReadBACK-Daten

Die Flagbits 5 bis 9 entstammen den letzten empfangenen ReadBACK-Statusworten. Wenn mindestens eine der 32 seriellen Schnittstellen ein gesetztes Bit ausweist, so wird dies hier angezeigt. Die auslesende Instanz kann dann über Registerinhalte des RemFPGA den genauen Status aller 32 Schnittstellen ermitteln und eine genauere Fehlersuche betreiben. Da während des normalen Betriebs in regelmäßigen Abständen ReadBACK-Daten ausgelesen werden, kann durch diese Headerinformationen ein ständiges Polling des gesamten RemFPGA Status vermieden werden. Stattdessen können weitere Aktionen gezielt nur bei Auftreten eines Fehlers durchgeführt werden.

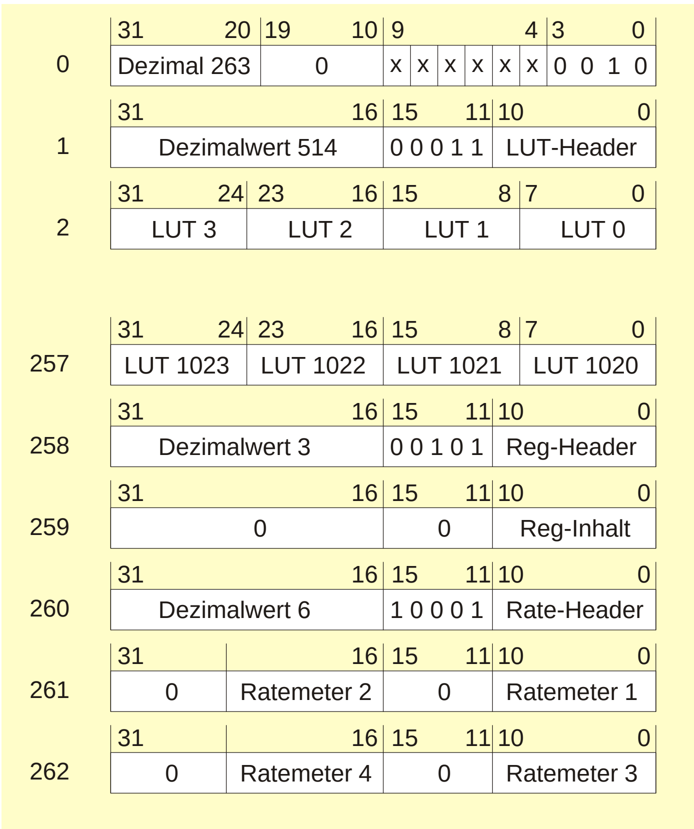


Abbildung 36 - ReadBACK Beispiel

Das resultierende Format des gesamten Datenblocks wird in einem Beispiel in Abbildung 36 gezeigt. Der RemFPGA hat Daten von drei verschiedenen Schnittstellen empfangen: Eine komplette Lookup-Tabelle von Schnittstelle Nummer 3, den Inhalt eines PPrASIC Registers über Schnittstelle 5 und eine Ratemeterinformation über Schnittstelle 17. Die Registerdaten benötigen nur drei 16 Bit Worte, daher wurde ein weiteres leeres 16 Bit Wort eingefügt. Damit beginnt der letzte Unterblock (Ratemeter) wie gefordert an einer 32 Bit Adresse.

6.3 Serielle Datenausgabe an die PPrASICs

Dieser Abschnitt beschreibt das Format, in welchem der RemFPGA Daten für die Ausgabe auf den seriellen Schnittstellen der PPrASICs erwartet.
Sowohl die VME-Schnittstelle wie auch der Pipelinebus verfügen über einen generalisierten Mechanismus zum Empfang von beliebigen Eingangsdaten. Erst eine Analyse des Headers dieser Daten teilt dem RemFPGA mit für welche interne Komponente sie bestimmt sind. Der RemFPGA in der vorliegenden Version akzeptiert Daten für die seriellen Schnittstellen, den I²C-Bus und den SPI-Bus. Die Unterscheidung erfolgt anhand des Wertes der ersten vier Bits des Datenstroms.
Ein Datenstrom für die seriellen Schnittstellen kann aus mehreren Unterblöcken bestehen. Jeder einzelne Unterblock kann gleichzeitig an eine beliebige Auswahl der 32 Schnittstellen gesendet werden.

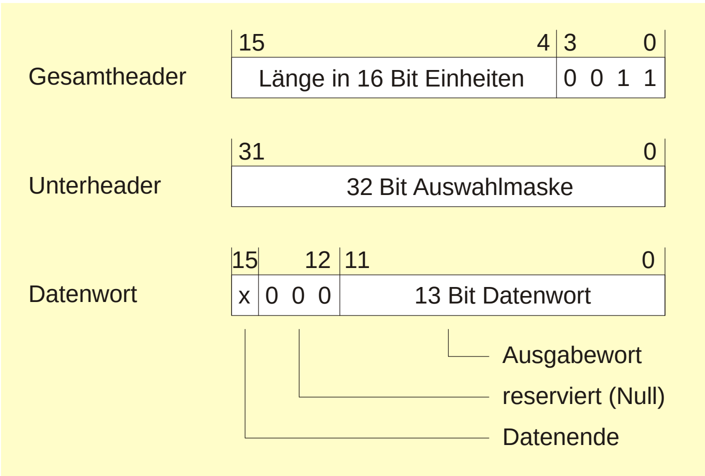


Abbildung 37 - Datenformat für serielle Ausgabe

Der Header des gesamten Eingabedatenstroms besitzt eine Breite von 16 Bit, der in den ersten vier Bits enthaltene ID-Code 3 lenkt diese Daten innerhalb des RemFPGA an das Modul RemSerOutControl.
Jeder Unterblock beginnt mit einer 32 Bit Auswahlmaske, worin jedes Bit eine serielle Schnittstelle repräsentiert. Die nachfolgenden Daten werden gleichzeitig an alle, durch ein gesetztes Bit ausgewählten Schnittstellen gesendet. Auf diese Weise ist es zum Beispiel möglich, bei allen 32 PPrASICs synchron die Datenverarbeitung zu aktivieren. Das Ende eines Unterblocks wird durch ein gesetztes höchstes Bit des letzten Datenworts angezeigt. Die einzelnen 13 Bit Datenworte werden ohne weitere Analyse oder Verarbeitung direkt an die PPrASICs gesendet.
Der interne Speicher des RemFPGA kann bis zu 4092 16 Bit Worte aufnehmen. Bei ausreichender verbleibender Kapazität können weitere Daten in den Speicher geschrieben werden noch während ältere Daten ausgeben werden.

6.4 Datenformat für die I²C-Schnittstelle

Eingangsdaten für die Ausgabe auf den I²C-Bus bestehen aus einen Header mit einer Länge von 16 Bit (siehe auch Abschnitt 6.3) und einer beliebigen Anzahl von 16 Bit Datenwörtern. Der Header trägt den ID-Code 4.

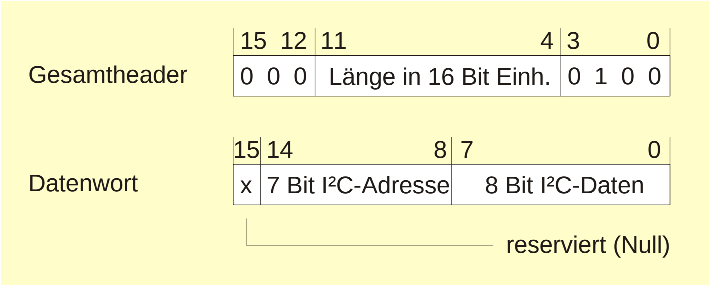


Abbildung 38 - Datenformat für I²C-Ausgabe

An den I²C-Bus werden voraussichtlich 32 Phos4-Chips angeschlossen, die jeweils mit einem einzigen I²C-Wort konfiguriert werden. Somit sind für eine vollständige Konfiguration 33 16 Bit Worte nötig. Der interne Speicher des RemFPGA kann bis zu 511 16 Bit Worte aufnehmen. Neue Daten werden erst nach vollständiger Verarbeitung der vorigen Daten akzeptiert.
Der I²C-Bus wird mit einer Geschwindigkeit von 95kBit/s betrieben.

6.5 Datenformat für die SPI-Schnittstellen

Die Eingangsdaten für die Ausgabe auf den vier SPI-Bussen bestehen (im Gegensatz zu I²C oder serieller Ausgabe) aus 32 Bit Wörtern. Der Header besitzt dieselbe Länge wie auch in Abschnitt 6.3 und in Abschnitt 6.4 beschrieben und verwendet den ID-Code 5. Er wird durch 16 weitere Bits (Null) auf die erforderliche Länge von 32 Bit gebracht.
An einen SPI-Bus sind im allgemeinen durch Hintereinanderschaltung mehrere Geräte angeschlossen. Im Falle des Präprozessormoduls sind vier MAX529 Digital-Analog-Wandler hintereinander geschaltet (auch „daisy-chaining" genannt). Die *Daisylänge* spezifiziert diese Anzahl und sollte daher im Falle der Präprozessormoduls immer auf vier gesetzt werden. Der Programmcode des RemFPGA (RemSpiControl) erfordert für alle vier SPI-Schnittstellen die gleiche *Daisylänge*.
Auf den 32 Bit Header folgen $4* < \mathit{Daisylänge} >$ 16 Bit Worte die an die vier SPI-Schnittstellen gesendet werden. Da es sich bei dem SPI-Bus um ein serielles Schiebeprotokoll handelt, landen die ersten vier Werte in den jeweils letzten angeschlossenen Geräten. Die letzten vier Werte werden zuletzt auf den Bus geschoben und werden somit in die ersten Geräte der vier Schnittstellen geladen.

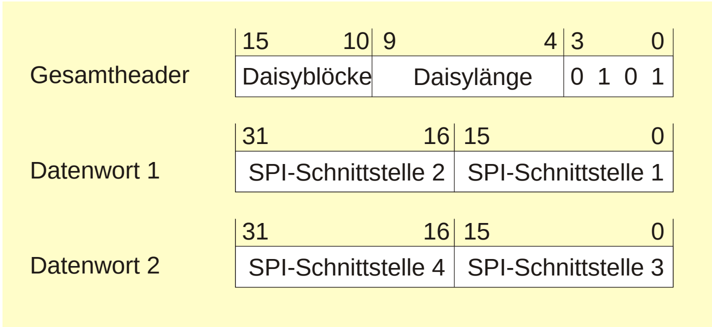


Abbildung 39 - Datenformat für SPI-Ausgabe

Der gesamte Datenblock kann aus mehreren dieser *Daisyblöcke* bestehen. Um alle MAX529 vollständig zu konfigurieren, sind neun Blöcke nötig, woraus sich eine Gesamtlänge von 73 32 Bit Worten ergibt.

Der interne Speicher des RemFPGA kann bis zu 255 32 Bit Wörter aufnehmen. Neue Daten werden erst nach vollständiger Verarbeitung der vorigen Daten akzeptiert.

Der SPI-Bus wird mit einer Geschwindigkeit von 1MBit/s betrieben.

6.6 Die Register des RemFPGA

Dieser Abschnitt erläutert die internen Register des RemFPGA. Sämtliche Register werden vom Modul RemRegControl verwaltet und können über VME als auch über den Pipelinebus angesprochen werden. Bei einem gleichzeitigen Schreibzugriff auf dasselbe Register über VME und Pipelinebus erhält die VME-Schnittstelle Vorrang, der Wert des Pipelinebusses wird ohne Fehlermeldung ignoriert.
Die Adresse der Register ist 7 Bit breit, die Register befinden sich im Adressbereich von ’h00 bis ’h7F[23]. Die Datenbreite eines jeden Registers beträgt 32 Bit. Für jedes Register ist in der Verilog Include-Datei IncRemGlobals eine Konstante mit dem entsprechenden Adresswert definiert. Die Spalte *Zugriff* gibt an, ob das entsprechende Register lesbar (*R*), schreibbar (*W*) oder beides (*R/W*) ist. Manche Register haben je nach Zugriffsart eine unterschiedliche Bedeutung, in diesem Fall sind zwei getrennte Tabelleneinträge für dasselbe Register vorhanden. Ist in der Spalte *Bedeutung* der Begriff *„bit-mask“* eingetragen, so steht jedes der 32 Bits dieses Registers für eine serielle Schnittstelle. Auf einige der Register wird auch in den folgenden beiden Abschnitten noch näher eingegangen.
Wird die Adresse eines nicht existierenden Registers ausgelesen, so wird der Wert 0 zugegeben, ein Schreibzugriff wird ohne Fehler ignoriert.

Addr.	Verilog Konstante	Zugriff	Bedeutung	
’h20	R_REG_RO_ACTIVE	R	bit-mask	Gesetzt, wenn SerInReadout momentan ReadOUT-Daten vom PPrASIC empfängt und verarbeitet.
’h21	R_REG_RO_ENABLE	R/W	bit-mask	Aktiviert die Datenverarbeitung in den RemSerInReadout Modulen.
’h22	R_REG_RO_DEBUG	R/W	bit-mask	Aktiviert den Debugmodus in den SerInReadout Modulen.
’h23	R_REG_RO_NOCOMPR	R/W	Bit 0	Deaktiviert die Kompression in allen SerInReadout Modulen (*vereinfachter Modus*, siehe Kapitel 5.3.3).
’h24	R_REG_RO_LOWMEM	R	bit-mask	ReadOUT Speicher fast erschöpft, SerInReadout speichert nur noch Header-Informationen.
’h25	R_REG_RO_OVERFLOW	R	bit-mask	ReadOUT Speicher völlig erschöpft, SerInReadout verwirft alle neuen Daten.
’h26	R_REG_RO_DATALEN	R	Bit 0-7	Gibt die Länge des nächsten auszule-senden ReadOUT-Datenblocks in 32 Bit Einheiten an.
’h30	R_REG_RB_ALERT	R	bit-mask	Gesetzt, wenn im entsprechenden ReadBACK-Statuswort mindestens ein Bit aktiviert ist.
’h31	R_REG_RB_ACTIVE	R	bit-mask	Gesetzt, wenn SerInReadback momen-tan ReadBACK-Daten vom PPrASIC empfängt.

[23] ’h bezeichnet in Verilog einen Hexadezimalwert, ’b definiert einen Binärwert

Addr.	Verilog Konstante	Zugriff	Bedeutung	
’h32	R_REG_RB_ENABLE	R/W	bit-mask	Aktiviert die Datenverarbeitung in den SerInReadback Modulen.
’h33	R_REG_RB_DATALOSS	R	bit-mask	ReadBACK-Daten wurden verworfen, da der Speicher erschöpft ist oder kein Zugriff auf den Speicher erfolgen konnte (siehe auch Kapitel 5.3.6).
’h34	R_REG_RB_MEMFULL	R	Bit 7-0	Speicher im entsprechenden der acht SerRbCollect Module ist erschöpft.
’h35	R_REG_RB_DATAREADY	R	Bit 7-0	Der Speicher im entsprechenden SerRbCollect Modul ist nicht leer.
			Bit 8-19	Gesamtanzahl an ReadBACK-Daten in allen acht SerRbCollect Modulen (in 32 Bit Einheiten). Eine folgende Auslese liefert diese Datenmenge.
			Bit 20	ReadBACK-Daten werden monentan ausgelesen. Bit 8-19 bleiben in dieser Phase konstant (siehe Kapitel 6.8)
			WICHTIG	Zur Entscheidung, ob ReadBACK-Daten zur Auslese bereit sind, sollte immer Bit 8-19 mit Null verglichen werden. Bit 7-0 sollten hierfür nicht verwendet werden!
’h40	R_REG_FPGA_STATUS	R	Bit 0	ReadOUT über PL-Bus[24] erlaubt
			Bit 1	ReadBACK über PL-Bus erlaubt
			Bit 2	Dateneingabe über PL-Bus erlaubt
			Bit 3	Register lesen über PL-Bus erlaubt
			Bit 4	Reg. schreiben über PL-Bus erlaubt
			Bit 5	ReadOUT über VME-Bus erlaubt
			Bit 6	ReadBACK über VME-Bus erlaubt
			Bit 7	Dateneingabe über VME-Bus erlaubt
			Bit 8	Datenausgabe über PL-Bus aktiv
			Bit 9	Dateneingabe über PL-Bus aktiv
			Bit 10	Datenausgabe über VME-Bus aktiv
			Bit 11	Dateneingabe über VME-Bus aktiv

[24] PL-Bus: Abkürzung für Pipelinebus

Addr.	Verilog Konstante	Zugriff	Bedeutung
’h41	R_REG_VME_INTRMASK	R/W	Dieses Register konfiguriert das Interrupt-Signal der „Easy VME“-Schnittstelle (LOW-aktiv) Bit 0 Interrupt, wenn ReadOUT-Daten verfügbar sind. Bit 1 Interrupt, wenn ReadBACK-Daten verfügbar sind. Bit 2 Interrupt, wenn keine serielle Ausgabe aktiv ist (SerOutControl Speicher ist leer). Bit 3 Interrupt, wenn keine Ausgabe über den I²C-Bus aktiv ist. Bit 4 Interrupt, wenn keine Ausgabe über den SPI-Bus aktiv ist. Bit 5 Interrupt, wenn über den PL-Bus kein Datentransfer erfolgt.
’h42	R_REG_VME_INTRFLAG	R	Die Bitbelegung ist mit der des obigen Registers identisch. Wenn ein Interrupt ausgelöst wurde, so ist/sind in diesem Register die Ursache/Ursachen abgelegt.
		W	Ein Schreibzugriff löscht das Interrupt-Signal (Der Datenwert des Schreibzugriffs wird ignoriert).

<table>
<tr><th>Addr.</th><th>Verilog Konstante</th><th>Zugriff</th><th colspan="2">Bedeutung</th></tr>
<tr><td rowspan="8">’h43</td><td rowspan="8">R_REG_IN_STATUS</td><td rowspan="8">R</td><td>Bit 0 – 11</td><td>freier Speicher für die serielle Ausgabe im SerOutControl Modul
(nur gültig, wenn Bit 12 gelöscht ist).</td></tr>
<tr><td>Bit 12</td><td>Serieller Speicher erschöpft.</td></tr>
<tr><td>Bit 13</td><td>Serielle Ausgabedaten wurden seit der letzten Auslese dieses Registers verworfen (Speicher war erschöpft).</td></tr>
<tr><td>Bit 14</td><td>Serielle Datenausgabe ist aktiv.</td></tr>
<tr><td>Bit 15</td><td>I²C-Daten wurden seit der letzten Auslese dieses Registers verworfen (Datenausgabe war noch aktiv).</td></tr>
<tr><td>Bit 16</td><td>I²C-Bus ist frei (keine Datenausgabe aktiv). Neue Daten können eingegeben und über I²C gesendet werden.</td></tr>
<tr><td>Bit 17</td><td>SPI-Daten wurden seit der letzten Auslese dieses Registers verworfen (Datenausgabe war noch aktiv).</td></tr>
<tr><td>Bit 18</td><td>SPI-Bus ist frei (keine Datenausgabe aktiv). Neue Daten können eingegeben und über SPI gesendet werden.</td></tr>
<tr><td rowspan="11">’h44</td><td rowspan="11">R_REG_DATA_STATUS</td><td rowspan="11">R</td><td colspan="2">Dieses Register enthält die wichtigsten Daten der vorigen Statusregister des RemFPGA.</td></tr>
<tr><td>Bit 0 – 11</td><td>ReadBACK Datenlänge (’h35)</td></tr>
<tr><td>Bit 12</td><td>ReadOUT Daten verfügbar (’h26)</td></tr>
<tr><td>Bit 13</td><td>mind. ein ReadOUT LowMem (’h24)</td></tr>
<tr><td>Bit 14</td><td>mind. ein ReadOUT Overflow (’h25)</td></tr>
<tr><td>Bit 15</td><td>mind. ein ReadBACK DataLoss (’h33)</td></tr>
<tr><td>Bit 16 – 27</td><td>freier serieller Ausgabespeicher (’h43)</td></tr>
<tr><td>Bit 28</td><td>serieller Ausgabespeicher voll (’h43)</td></tr>
<tr><td>Bit 29</td><td>serielle Ausgabedaten verworfen (’h43)
(Flag wird bei Auslese nicht gelöscht)</td></tr>
<tr><td>Bit 30</td><td>I²C-Bus ist frei. (’h43)</td></tr>
<tr><td>Bit 31</td><td>SPI-Bus ist frei. (’h43)</td></tr>
<tr><td rowspan="2">’h45</td><td rowspan="2">R_REG_PIPE_ADDR</td><td rowspan="2">R</td><td>Bit 0 – 5</td><td>Pipelinebus Node Adresse</td></tr>
<tr><td>Bit 8 – 16</td><td>Pipelinebus Node Gruppenadresse</td></tr>
<tr><td rowspan="6">’h46</td><td rowspan="6">R_REG_SIGNALS</td><td rowspan="6">W</td><td>Bit 0</td><td>Sende Reset an ReadOUT Module</td></tr>
<tr><td>Bit 1</td><td>Sende Reset an ReadBACK Module</td></tr>
<tr><td>Bit 2</td><td>Sende Reset an SerOutControl Modul</td></tr>
<tr><td>Bit 3</td><td>Sende Reset an I²C Module</td></tr>
<tr><td>Bit 4</td><td>Sende Reset an SPI Module</td></tr>
<tr><td>Bit 5</td><td>Softreset des gesamten RemFPGA</td></tr>
</table>

Addr.	Verilog Konstante	Zugriff	Bedeutung	
’h50	R_REG_SAMPLES0117	R/W	Bit 0 – 4	Anzahl zu verarbeitender BCID-Werte des ersten Kanals der Schnittstelle 01.
’h51	R_REG_SAMPLES0218		Bit 5 – 7	Anzahl zu verarbeitender RAW-Werte des ersten Kanals der Schnittstelle 01.
’h53	R_REG_SAMPLES0319		Bit 8 – 12	Anzahl zu verarbeitender BCID-Werte des zweiten Kanals der Schnittstelle 01.
…	…		Bit 13 – 15	Anzahl zu verarbeitender RAW-Werte des zweiten Kanals der Schnittstelle 01.
			Bit 16 – 20	Anzahl zu verarbeitender BCID-Werte des ersten Kanals der Schnittstelle 17.
’h5D	R_REG_SAMPLES1430		Bit 21 – 23	Anzahl zu verarbeitender RAW-Werte des ersten Kanals der Schnittstelle 17.
’h5E	R_REG_SAMPLES1531		Bit 24 – 28	Anzahl zu verarbeitender BCID-Werte des zweiten Kanals der Schnittstelle 17.
’h5F	R_REG_SAMPLES1632		Bit 29 – 31	Anzahl zu verarbeitender RAW-Werte des zweiten Kanals der Schnittstelle 17.
’h60 … ’h7F	R_REG_RB_STATUS01 … R_REG_RB_STATUS32	R	Bit 0 – 9	letztes empfangenes ReadBACK-Statuswort der entsprechenden seriellen Schnittstelle

Tabelle 6 - Register des RemFPGA

6.7 Die Verwendung des Pipelinebusses

Die Grundzüge des Pipelinebusses werden in Kapitel 3.3.5 und in [8] dargelegt. In diesem Abschnitt werden die einzelnen Pipelinebus Befehle des RemFPGA erläutert und die Vorgehensweise zur Dateneingabe und zur Datenauslese beschrieben.

6.7.1 Die Befehle des Pipelinebusses

Tabelle 7 führt alle Befehle des Pipelinebusses auf. Ein Befehl enthält neben dem Befehlscode (dem so genannten *Token*) und der Zieladresse auch ein 16 Bit Argument. Die Bedeutung dieses Arguments ist in der Tabelle bei Bedarf ebenfalls aufgeführt. Soweit möglich, wurden dieselben Befehle und Befehlscodes verwendet wie von dem Prototyp des RODs (siehe [8]). Einige Befehle sind jedoch neu während andere nicht mehr existieren. Die Befehlscodes von veralteten Befehlen wurden nicht wieder verwendet, wodurch Lücken im Nummerierungsschema der Codes entstehen. Jedem Befehlscode ist eine Verilog Konstante zugeordnet, welche in der Include-Datei IncRemPLBus definiert ist.

Token	Konstante	Bedeutung	
’h00	PBT_NOP	Dummy-Befehl, es wird keine Aktion ausgeführt	
’h01	PBT_LoadNodeAddress	Bit 0 – 5	Das Argument wird als neue Pipelinebus Node Adresse übernommen, der Wert wird um eins erhöht und weitergeschickt (siehe Abschnitt 6.7.2).
’h02	PBT_StartInput	Dateneingabe über PL-Bus wird aktiviert (siehe Abschnitt 6.7.4)	
’h03	PBT_StartReadout	PL-Bus ReadOUT Auslese wird aktiviert (siehe Abschnitt 6.7.3)	
’h0A	PBT_GetStatus	Bit 0 – 15	Inhalt des Registers FPGA_STATUS wird im Argument abgelegt.
’h0E	PBT_SendSignal	Bit 0 – 15	Inhalt des Arguments wird in das Register SIGNALS übertragen.
’h0F	PBT_BeginOfData	Beginnt einen Datenblock (siehe Kapitel 3.3.5)	
’h10	PBT_EndOfData	Beendet einen Datenblock (siehe Kapitel 3.3.5)	
’h11	PBT_LoadGroupAddress	Bit 0 – 5	Konfiguriert die Node Gruppenadresse (siehe 6.7.2)
’h12	PBT_GetAddress	Bit 0 – 5 Bit 8 – 13	Enthält die Node Adresse Enthält die Node Gruppenadresse
’h13	PBT_StartReadback	PL-Bus ReadBACK Auslese wird aktiviert (siehe Abschnitt 6.7.3)	
’h14	PBT_StopData	Beendet Dateneingabe/-ausgabe über den PL-Bus	
’h15	PBT_GetDataOutInfo	Bit 0 – 15	Inhalt der unteren 16 Bit des Registers DATA_STATUS wird im Argument abgelegt.
’h16	PBT_GetDataInInfo	Bit 0 – 15	Inhalt der oberen 16 Bit des Registers DATA_STATUS wird im Argument abgelegt.
’h17	PBT_StartReadRegister	Auslese von Registern über PL-Bus wird aktiviert (siehe 6.7.3). Bit 0 – 6 Bit 8 – 14	 Register Startadresse Anzahl auszulesende Register

Token	Konstante	Bedeutung
’h18	PBT_StartWriteRegister	Auslese von Registern über PL-Bus wird aktiviert (siehe 6.7.4). Bit 0 – 6 Register Startadresse

Tabelle 7 - Befehle des Pipelinebusses

Ein Pipelinebus Befehl wird über den Binärwert ’b10 in den beiden höchstwertigen Bits eines 34 Bit Pipelinebus-Wortes identifiziert (die Parity-Information in Bit 35 nicht mitgerechnet). Abbildung 40 zeigt die Bitbelegung eines solchen Befehls.
Schickt der Pipelinebus Master einen Befehl auf den Pipelinebus, so löscht er sowohl das *Accept Flag* als auch das *Error Flag*. Das *Adressfeld* kann entweder mit der Adresse einer bestimmten Slave-Node belegt werden oder auch mit einer Gruppenadresse (siehe auch Abschnitt 6.7.2). Empfängt und erkennt eine adressierte Slave-Node einen Befehl, so führt er die entsprechende Aktion aus (wobei unter Umständen das Befehlsargument verändert wird) und reicht das Befehlswort mit gesetztem *Accept Flag* weiter an die nächste Node. Kommt das Befehlswort nach einem vollen Umlauf wieder beim Pipelinebus-Master an, so kann dieser erkennt, ob mindestens eine Slave-Node den Befehl verarbeitet hat.

Fehlerworte des Pipelinebusses

Kann eine Slave-Node einen Befehl nicht verarbeiten oder erkennt eine Node einen Parity-Fehler, so wird das fehlerhafte Wort durch ein Befehlswort mit gesetztem *Error Flag* ersetzt. Die Slave-Node trägt in das *Adressfeld* seine eigene Node Adresse ein und füllt das Argument mit einem Fehlerwert. Der Inhalt des *Befehlsfeldes* bleibt unverändert. Die Bedeutung der verschiedenen Bits wird in den folgenden Abschnitten näherer erläutert.

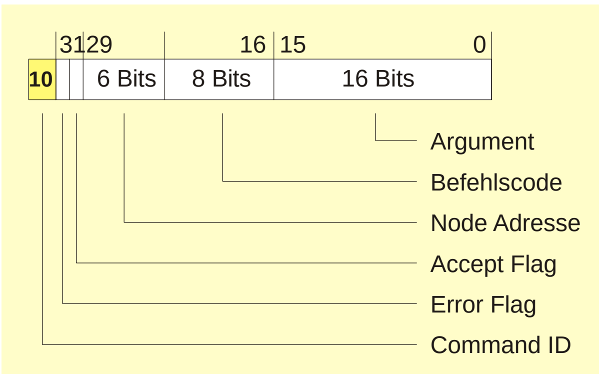


Abbildung 40 - Format eines Pipelinebus Befehls

Bit	Bedeutung
0	Parityfehler entdeckt
1	Fehler wegen aktiver Datenausgabe.
2	nicht verwendet
3	Eingabedaten konnten nicht schnell genug verarbeitet werden.
4	nicht verwendet
5	Falsche Datenübertragungsrichtung
6	nicht verwendet
7	Fehler wegen aktiver Dateneingabe.
8	Unbekannter Befehlscode
9	nicht verwendet
10	Befehlscode nicht zulässig
11 – 15	nicht verwendet

Tabelle 8 - Fehlerwort des Pipelinebusses
(für Definitionen von Konstanten für
die Fehlerbits siehe IncRemPLBus)

6.7.2 Konfiguration der Pipelinebus Node Adresse

Nach einem Systemneustart ist jede Slave-Node mit der Adresse ’h3F und der Gruppenadresse ’h3E konfiguriert. Die erste Aufgabe des Masters ist die Detektion der Anzahl angeschlossener Slave-Nodes und die Konfiguration der Adressen aller Slaves.
Dies geschieht durch die Verwendung des Befehls *LoadNodeAddress*. Der Master füllt das Adressfeld mit dem Wert ’h3F und Argument mit dem Wert 0. Da alle Slaves auf diese Adresse reagiert, konfiguriert sich der erste Slave mit der Node Adresse 0. Er inkrementiert sodann den Wert des Arguments und schickt den Befehl an die zweite Node. Diese konfiguriert sich daher mit der Node Adresse 1. Nach einem vollen Umlauf sind alle Node mit von Null ansteigenden Adressen konfiguriert und der ursprüngliche *LoadNodeAddress* Befehl erreicht den Master mit der Anzahl der angeschlossenen Slave-Nodes im Argument.
Der Adresse ’h3F kommt auch nach dieser Konfiguration eine besondere Bedeutung zu – sie ist als Broadcast-Adresse definiert und jeder angeschlossene Slave verarbeitet einen Befehl mit dieser Adresse. Anschließend kann jeder Node mit dem Befehl *LoadGroupAddress* eine zusätzliche Gruppenadresse zugeordnet werden. Jede Node reagiert somit auf seine Node Adresse, seine Gruppenadresse und die Broadcast-Adresse ’h3F.

6.7.3 Datenauslese über den Pipelinebus

Über den Pipelinebus können drei verschiedene Arten von Daten ausgelesen werden: ReadOUT-Daten, ReadBACK-Daten und RemFPGA Register. Die grundsätzliche Art der Datenübertragung mit Hilfe von *BeginOfData*/*EndOfData* Befehlen ist in allen drei Fällen identisch und in Kapitel 3.3.5 erläutert. Dieser Abschnitt erläutert den für einen Ausgabezyklus nötigen Konfigurationsvorgang.

Für jeden der drei Datentypen existiert ein entsprechender Pipelinebus *Start*-Befehl (*Start-Readout*, *StartReadback* und *StartReadRegister*). Bei Empfang eines solchen Befehls geht der RemFPGA in den Bereitschaftszustand für die Ausgabe der gewünschten Daten. Ein *Start*-Befehl wird jedoch nur akzeptiert, wenn sich der RemFPGA nicht schon im Wartezustand für die Verarbeitung einer anderen Datenart befindet. Sollte dies der Fall sein, so lehnt der RemFPGA den Befehl ab und erzeugt ein Fehlerwort mit gesetztem Bit 10 (Befehlscode nicht zulässig). Im Falle von ReadOUT oder ReadBACK wird der Befehl auch abgelehnt, falls diese Datenart in diesem Moment über den VME-Bus ausgelesen wird. Wird der Befehl jedoch erfolgreich verarbeitet, so setzt der RemFPGA das *Accept Flag* und geht in Ausgabebereitschaft. Um Fehler zu vermeiden, sollte der Master über den Befehl *GetStatus* die momentan möglichen Ausgabemodi erfragen und entsprechend reagieren (siehe auch Abschnitt 6.7.5).
Befindet sich der RemFPGA in Ausgabebereitschaft, so werden wie erwähnt andere *Start*-Befehl abgelehnt, andere Befehle wie zum Beispiel *GetDataOutInfo* werden jedoch weiterhin verarbeitet. Empfängt der RemFPGA einen *BeginOfData*-Befehl, so werden folgende leere Wörter mit den entsprechenden Daten gefüllt (siehe Kapitel 3.3.5, Seite 34). Nach Abschluss der Ausgabe fügt der RemFPGA einen *EndOfData*-Befehl ein und geht wieder in Ausgabebereitschaft. Diese bedeutet, dass für einen weiteren Ausgabezyklus derselben Datenart kein erneuter *Start*-Befehl nötig ist. Dies hat auch den (erwünschten) Nebeneffekt, dass der VME-Bus während dieser Phase komplett für die Ausgabe dieser Datenart gesperrt bleibt.
Um die Bereitschaftsphase zu verlassen, sendet der Master den Befehl *StopData*, danach werden wieder beliebige neue *Start*-Befehle akzeptiert.
Befindet sich der RemFPGA nicht mehr nur in Bereitschaft, sondern ist aktiv mit der Datenausgabe beschäftigt, so werden sämtliche *Start*-Befehle mit einem Fehler (Bit 1 - Fehler wegen aktiver Datenausgabe) quittiert. Während dieser Phase dürfen auch keine Datenwörter über den Pipelinebus empfangen werden, da der RemFPGA selbst Datenwörter auf dem Bus ausgibt. Der Empfang eines Datenwortes wird daher ebenfalls mit diesem Fehlerwort markiert.

6.7.4 Dateneingabe über den Pipelinebus

Über den Pipelinebus können zwei verschiedene Arten von Daten eingegeben werden: so genannte Eingabe-Daten und RemFPGA Register. Die grundsätzliche Art der Datenübertragung mit Hilfe von *BeginOfData*/*EndOfData* Befehlen ist in beiden Fällen identisch und in Kapitel 3.3.5 erläutert. Dieser Abschnitt erläutert den für einen Eingabezyklus nötigen Konfigurationsvorgang.
Ein Eingabedatenblock (eingeschlossen von Begin/End) kann Daten für die Ausgabe über die seriellen Schnittstellen, den I²C-Bus oder die SPI-Schnittstellen enthalten. Der Pipelinebus selbst unterscheidet diese Untertypen jedoch nicht, eine Zuordnung erfolgt durch andere Programmteile des RemFPGA.
Analog zur Datenausgabe existieren auch hier entsprechende *Start*-Befehle (*StartInput* und *StartWriteRegister*). Das Vorgehen ist analog zur Ausgabe, auch hier werden *Start*-Befehle abgelehnt, falls eine Eingabe über den VME-Bus aktiv sein sollte oder der Pipelinebus gerade für Datenausgabe konfiguriert sein sollte. Daher sollte der Master zuvor den Befehl *GetStatus* verwenden.

Befindet sich der RemFPGA in Eingabebereitschaft, so wird eine beliebige Anzahl von Datenblöcken (eingeschlossen durch *BeginOfData/EndOfData*) vom RemFPGA entgegen genommen und verarbeitet. Da der Pipelinebus etwas langsamer getaktet sein wird als der RemFPGA selbst, können eingehende Daten mit ausreichender Geschwindigkeit verarbeitet werden. Sollte dies in Extremsituationen nicht der Fall sein, so wird das deshalb unverarbeitete Datenwort durch ein Fehlerwort (Bit 3 - Eingabedaten konnten nicht schnell genug verarbeitet werden) ersetzt und die Verarbeitung des momentanen Datenblocks wird abgebrochen. Schon empfangende Daten dieses unvollständigen Blocks werden verworfen. Der RemFPGA geht zurück in Eingabebereitschaft. Dem Autor dieser Diplomarbeit sind keine Fälle bekannt, in denen eine solche Situation auftreten könnte und sie ist in Simulationen nicht aufgetreten. Sollte dies jedoch trotzdem vorkommen, so sollte der Pipelinebus-Master den abgebrochenen Datenblock erneut senden und dabei nach jedem zweiten Datenwort einen *NOP*-Befehl einfügen. Nach Abschluss aller Eingabeblöcke wird der Bereitschaftszustand durch den Befehl *StopData* verlassen.

6.7.5 Besonderheiten und Anwendungsbeispiel

Die Verwendung verschiedener Signale für die interne Taktung des RemFPGA und den Betrieb des Pipelinebusses bedingt einige Besonderheiten der internen Verarbeitung von Pipelinebus-Befehlen durch den RemFPGA:

- Zwei an denselben RemFPGA adressierte Pipelinebus-Befehle dürfen niemals direkt nacheinander folgen. Um eine direkte Folge zu vermeiden, können bei Bedarf *NOP*-Befehle eingefügt werden, welche nicht dieser Beschränkung unterliegen.
- Bei der Datenauslese kann es hin und wieder zu Synchronisationsschwierigkeiten zwischen den beiden Taktsignalen kommen. In diesem Fall fügt der RemFPGA zwischen einzelnen Datenwörtern einen *NOP*-Befehl ein. Der ROD sollte daher *NOP*-Befehle innerhalb eines BeginOfData/EndOfData Blocks grundsätzlich ignorieren.

Die folgenden Beispiele veranschaulichen die Anwendung des Pipelinebusses. Dabei wird zur Vereinfachung von einem Bus mit einem Master und zwei Slaves ausgegangen. Nach drei Taktzyklen empfängt also der Master ein von im ausgegebenes Pipelinebus-Wort wieder auf seinem Eingang. Der erste Slave empfängt ein Master-Wort nach einem Taktzyklus, der zweite Slave nach zwei Zyklen.

Auslese von ReadOUT Daten

Die folgende Tabelle 9 listet die einzelnen Aktionen der drei beteiligten Pipelinebus Nodes. Dabei wird davon ausgegangen, dass die Slaves sich im Ruhezustand befinden und keine Fehler verursachen. Wartet der Master auf Reaktionen der Slaves, so sendet er in dieser Zeit leere Wörter. In diesem Beispiel sendet der erste Slave zwei Datenwörter und der zweite Slave drei Wörter. Tabelle 10 zeigt die resultierenden Datenströme auf dem Ausgang und dem Eingang des Pipelinebus-Masters.

<table>
<tr><th>Takt</th><th>Aktion des Masters</th><th>Aktion von Slave 0</th><th>Aktion von Slave 1</th></tr>
<tr><td>1</td><td>sendet StartReadout an ’h3F</td><td>im Ruhezustand</td><td rowspan="2">im Ruhezustand</td></tr>
<tr><td>2</td><td rowspan="2">wartet auf Bestätigung</td><td>geht in Ausgabebereitschaft</td></tr>
<tr><td>3</td><td rowspan="3">wartet auf BeginOfData</td><td>geht in Ausgabebereitschaft</td></tr>
<tr><td>4</td><td>empfängt StartReadout</td><td rowspan="2">wartet auf BeginOfData</td></tr>
<tr><td>5</td><td>sendet BeginOfData an ’h3F</td></tr>
</table>

Takt	Aktion des Masters	Aktion von Slave 0	Aktion von Slave 1
6	wartet auf Datenwörter und *EndOfData* (und sendet leere Wörter)	geht in aktiven Modus	
7		sendet Datenwort A1	geht in aktiven Modus
8		sendet Datenwort A2	wartet auf leere Wörter
9	empfängt Datenwort A1	sendet *EndOfData* (Addr. 0)	
10	empfängt Datenwort A2	wartet auf *BeginOfData*	
11	empfängt *EndOfData* 0		sendet Datenwort B1
12	empfängt Datenwort B1		sendet Datenwort B2
13	empfängt Datenwort B2		sendet Datenwort B3
14	empfängt Datenwort B3		sendet *EndOfData* (Addr. 1)
15	empfängt *EndOfData* 1		wartet auf *BeginOfData*

Tabelle 9 - Pipelinebus ReadOUT Beispiel

Takt	Ausgabe	Empfang
1	*StartReadout* an ’h3F	unbekannt/unbedeutend
2	*leeres Wort*	unbekannt/unbedeutend
3	*leeres Wort*	unbekannt/unbedeutend
4	*leeres Wort*	*StartReadout* mit Accept Flag
5	*BeginOfData* an ’h3F	*leeres Wort*
6	*leeres Wort*	*leeres Wort*
7	*leeres Wort*	*leeres Wort*
8	*leeres Wort*	*BeginOfData* mit Accept Flag
9	*leeres Wort*	Datenwort A1
10	*leeres Wort*	Datenwort A2
11	*leeres Wort*	*EndOfData* von ’h00
12	*leeres Wort*	Datenwort B1
13	*leeres Wort*	Datenwort B2
14	*leeres Wort*	Datenwort B3
15	*leeres Wort*	*EndOfData* von ’h01

Tabelle 10 - Pipelinebus ReadOUT Beispiel

Setzen von RemFPGA Registern

In diesem Beispiel werden die Register ’h21 bis ’h23 des ersten Slaves mit neuen Werten beschrieben. Anschließend werden die Register ’h41 und ’h42 des zweiten Slaves gesetzt. Der zweite Slave sei jedoch schon für die Ausgabe von ReadBACK-Daten konfiguriert. Siehe Tabelle 11 und Tabelle 12.

<table>
<tr><th>Takt</th><th>Aktion des Masters</th><th>Aktion von Slave 0</th><th>Aktion von Slave 1</th></tr>
<tr><td>1</td><td>sendet StartWriteRegister an ’h00 mit Argument ’h21</td><td>im Ruhezustand</td><td rowspan="14">in Ausgabebereitschaft für ReadBACK-Daten

in Ausgabebereitschaft für ReadBACK-Daten</td></tr>
<tr><td>2</td><td rowspan="2">wartet auf Bestätigung</td><td>geht in Eingabebereitschaft</td></tr>
<tr><td>3</td><td rowspan="3">wartet auf BeginOfData</td></tr>
<tr><td>4</td><td>empfängt StartWriteRegister</td></tr>
<tr><td>5</td><td>sendet BeginOfData an ’h00</td></tr>
<tr><td>6</td><td>sendet Datenwort Reg. ’h21</td><td>geht in aktiven Modus</td></tr>
<tr><td>7</td><td>sendet Datenwort Reg. ’h22</td><td>setzt Register ’h21</td></tr>
<tr><td>8</td><td>sendet Datenwort Reg. ’h23</td><td>setzt Register ’h22</td></tr>
<tr><td>9</td><td>sendet EndOfData an ’h00</td><td>setzt Register ’h23</td></tr>
<tr><td>10</td><td rowspan="2">wartet auf EndOfData und sendet leere Wörter</td><td>empfängt EndOfData</td></tr>
<tr><td>11</td><td rowspan="3">wartet auf BeginOfData</td></tr>
<tr><td>12</td><td>empfängt EndOfData</td></tr>
<tr><td>13</td><td>sendet StopData an ’h00</td></tr>
<tr><td>14</td><td>sendet StartWriteRegister an ’h01 mit Argument ’h41</td><td>geht in Ruhezustand</td></tr>
<tr><td>15</td><td rowspan="2">wartet auf Bestätigung</td><td rowspan="4">im Ruhezustand</td><td>wartet auf BeginOfData</td></tr>
<tr><td>16</td><td>empfängt StartWriteRegister</td></tr>
<tr><td>17</td><td>empfängt Fehlerwort</td><td>sendet Fehlerwort (Bit 10)</td></tr>
<tr><td>18</td><td>verarbeitet Fehlerzustand</td><td>in Ausgabebereitschaft für ReadBACK-Daten</td></tr>
</table>

Tabelle 11 - Pipelinebus Register Beispiel

Takt	Ausgabe	Empfang
1	*StartWriteRegister* an ’h00 (Arg. ’h21)	unbekannt/unbedeutend
2	*leeres Wort*	unbekannt/unbedeutend
3	*leeres Wort*	unbekannt/unbedeutend
4	*leeres Wort*	*StartWriteRegister* mit Accept von ’h00
5	*BeginOfData* an ’h00	*leeres Wort*
6	Datenwort Register ’h21	*leeres Wort*
7	Datenwort Register ’h22	*leeres Wort*
8	Datenwort Register ’h23	*BeginOfData* mit Accept von ’h00
9	*EndOfData* an ’h00	Datenwort Register ’h21
10	*leeres Wort*	Datenwort Register ’h22
11	*leeres Wort*	Datenwort Register ’h23
12	*leeres Wort*	*EndOfData* mit Accept von ’h00
13	*StopData* an ’h00	*leeres Wort*

Takt	Ausgabe	Empfang
14	*StartWriteRegister* an ’h01 (Arg. ’h41)	*leeres Wort*
15	*leeres Wort*	*leeres Wort*
16	*leeres Wort*	*StopData* mit Accept von ’h00
17	*leeres Wort*	*Fehlerwort* mit Bit 10 von ’h01
18	Reaktion auf Fehler	*leeres Wort*

Tabelle 12 - Pipelinebus Register Beispiel

6.8 Verwendung der VME-Schnittstelle

Die Signalabfolge zur Kommunikation über die „Easy VME“-Schnittstelle ist in Kapitel 3.3.4 dargelegt. Dieser Abschnitt soll dagegen Auskunft geben über das Adressierungsschema der VME-Implemetation des RemFPGA geben. Grundsätzlich kann über die 22 Adressleitungen des RemFPGA ein Adressraum von 4 MB mit einer Datenbreite von 32 Bit angesteuert werden. Der RemFPGA macht jedoch nur von einem kleinen Bruchteil dieser Kapazität gebrauch. Lediglich die unteren 10 Adressbits werden verwendet und belegen so den Adressraum von ’h000 bis ’h3FF. Von diesem kleinen Bereich ist nur etwa ein Achtel mit Funktionen belegt.
Der Bereich von ’h100 bis ’h17F erlaubt den direkten Zugriff auf die internen Register des RemFPGA. VME-Bus und Pipelinebus können gleichzeitig auf die Register zugreifen, sollte ein gleichzeitiger Schreibzugriff auf dasselbe Register erfolgen, so hat der Wert des VME-Busses Vorrang, der Wert des Pipelinebusses wird ohne Fehlermeldung verworfen.
Zusätzlich dienen sechs weitere Adressen zur Abwicklung der Dateneingabe und Datenausgabe. Diese sind in Tabelle 13 aufgeführt und werden im folgenden Text weiter erläutert. Auch für diese Adressen sind Verilog Konstanten in der Include-Datei IncRemGlobals definiert.

<table>
<tr><th>Addr.</th><th>Konstante</th><th>Zugriff</th><th colspan="2">Bedeutung</th></tr>
<tr><td>’h200</td><td>R_VME_READFIFO</td><td>R</td><td colspan="2">Zugriff auf das nächste Ausgabewort</td></tr>
<tr><td>’h201</td><td>R_VME_WRITEFIFO</td><td>W</td><td colspan="2">Zugriff auf das nächste Eingabewort</td></tr>
<tr><td rowspan="7">’h210</td><td rowspan="7">R_VME_FIFOSTATUS</td><td rowspan="7">R</td><td colspan="2">Status der beiden FIFO Elemente</td></tr>
<tr><td>Bit 0</td><td>ReadOUT über FIFO aktiv</td></tr>
<tr><td>Bit 1</td><td>ReadBACK über FIFO aktiv</td></tr>
<tr><td>Bit 2</td><td>Eingabe über FIFO aktiv</td></tr>
<tr><td>Bit 16</td><td>ReadOUT über FIFO erlaubt</td></tr>
<tr><td>Bit 17</td><td>ReadBACK über FIFO erlaubt</td></tr>
<tr><td>Bit 18</td><td>Eingabe über FIFO erlaubt</td></tr>
<tr><td rowspan="7">’h211</td><td rowspan="7">R_VME_STARTREADOUT</td><td colspan="3">Steuerung der ReadOUT Auslese über VME</td></tr>
<tr><td rowspan="4">R</td><td>Bit 0 – 15</td><td>Anzahl ReadOUT-Daten</td></tr>
<tr><td>Bit 16</td><td>ReadOUT-Zyklus wurde initiiert</td></tr>
<tr><td>Bit 17</td><td>ReadOUT-Zyklus ist aktiv</td></tr>
<tr><td>Bit 18</td><td>FIFO enthält letztes Wort</td></tr>
<tr><td rowspan="2">W</td><td>Nullwert</td><td>Startet einen neuen ReadOUT-Zyklus</td></tr>
<tr><td>nicht-Null</td><td>Bricht einen ReadOUT-Zyklus ab</td></tr>
</table>

<table>
<tr><th>Addr.</th><th>Konstante</th><th>Zugriff</th><th colspan="2">Bedeutung</th></tr>
<tr><td>’h212</td><td>R_VME_STARTREADBACK</td><td>R/W</td><td colspan="2">Steuerung der ReadBACK Auslese über VME
(Funktion identisch zu STARTREADOUT)</td></tr>
<tr><td rowspan="7">’h213</td><td rowspan="7">R_VME_STARTINPUT</td><td rowspan="5">R</td><td colspan="2">Steuerung der Dateneingabe über VME</td></tr>
<tr><td>Bit 0 – 15</td><td>Anzahl empfangender Daten</td></tr>
<tr><td>Bit 16</td><td>Eingabezyklus wurde initiiert</td></tr>
<tr><td>Bit 17</td><td>Eingabezyklus ist aktiv</td></tr>
<tr><td>Bit 18</td><td>Datenüberlauf, Abbruch erforderlich</td></tr>
<tr><td rowspan="2">W</td><td>Nullwert</td><td>Startet einen neuen Eingabezyklus</td></tr>
<tr><td>nicht-Null</td><td>Beendet einen Eingabezyklus</td></tr>
</table>

Tabelle 13 - Datentransfer über VME

6.8.1 Datenauslese über die VME-Schnittstelle

Über den VME-Bus können zwei verschiedene Datentypen ausgelesen werden: ReadOUT-Daten und ReadBACK-Daten. Der Datentransfer erfolgt mit der Hilfe eines so genannten FIFO-Mechanismus[25]. Für beide Datentypen ist der FIFO an der Adresse READFIFO zuständig. Über die beiden Adressen STARTREADOUT und STARTREADBACK wird der Datentransfer gesteuert.

Im Ruhezustand liefert ein Lesezugriff auf den FIFO den Wert 0. Um einen Auslesezyklus zu starten, wird der Wert 0 an die Adresse STARTREADOUT oder STARTREADBACK geschrieben. Der RemFPGA geht daraufhin in Ausgabebereitschaft. Dies ist jedoch nur möglich, wenn nicht schon ein anderer Ausgabezyklus aktiv ist und wenn der angeforderte Datentyp nicht in diesem Moment für die Ausgabe über den Pipelinebus konfiguriert ist. Der Inhalt von FIFOSTATUS gibt Auskunft über den momentanen Zustand der VME-FIFOs.

Nach Schreibzugriff mit dem Wert 0 auf die FIFO-Steuerung (ReadBACK oder ReadOUT) liefert ein darauf folgender Lesezugriff das Resultat des Startversuchs. Bei Erfolg sind die Bits 16 und 17 gesetzt und die Anzahl auszulesender Daten wird im niederwertigen Wort zurückgeliefert. Sequentielle Lesezugriffe auf den Auslese-FIFO liefern den angeforderten Datenblock. Befindet sich im FIFO das letzte Wort des auszulesenden Datenblocks, so ist Bit 18 in der FIFO-Steuerung gesetzt. Bit 16 ist lediglich direkt nach dem Start eines Zyklus vor der Auslese des ersten Wortes aus dem FIFO gesetzt. Danach ist lediglich Bit 17 gesetzt, bis dieses nach der Auslese des letzten Wortes ebenfalls gelöscht wird. Die Auslese des letzten FIFO-Wortes beendet den Zyklus und der RemFPGA geht wieder in den Ruhezustand über.

Dies bedeutet, dass sofort nach der Auslese des letzten Wortes der gerade ausgelesene Datentyp auch wieder über den Pipelinebus angefordert werden kann. Im Unterschied dazu geht der Pipelinebus nicht nach einem *EndOfData* wieder in den Ruhezustand, sondern erst nach Empfang eines *StopData*.

Grundsätzlich ist es irrelevant, ob eine Ausleseoperation die Anzahl auszulesender Worte aus der FIFO-Steuerung bezieht und verwendet oder ob immer abwechselnd Auslese-FIFO und FIFO-Steuerung ausgelesen werden, bis schliesslich in der FIFO-Steuerung Bit 18 gesetzt wird. (Woraufhin ein weiterer letzter Zugriff auf den Auslese-FIFO zu erfolgen hat).

Sollte der Auslesezyklus durch einen Schreibzugriff auf die FIFO-Steuerung mit einem Wert ungleich Null abgebrochen werden, so verwirft der RemFPGA die restlichen Auslesedaten in seinem internen Speicher und geht in den Ruhezustand über.

[25] FIFO – First-In-First-Out

6.8.2 Dateneingabe über die VME-Schnittstelle

Die Dateneingabe über VME erfolgt fast vollständig analog zur Datenauslese. In diesem Fall wird der Eingabe-FIFO an der Adresse WRITEFIFO verwendet. Die VME-Schnittstelle des RemFPGA behandelt alle Eingabedaten gleich, die Unterscheidung von seriellen Daten, I²C-Daten und SPI-Daten erfolgt in anderen Teilen des RemFPGA.

Der Start und der Abbruch eines Eingabezyklus erfolgt absolut identisch zur Datenauslese. Nach Start der Operation werden die Eingabedaten sequentiell in den Eingabe-FIFO geschrieben. Nach der Übertragung des letzten Wortes wird der Zyklus durch das Schreiben eines Wertes ungleich Null in die FIFO-Steuerung beendet. Daraufhin geht der RemFPGA wieder in den Ruhezustand. Vor der Beendigung eines Eingabezyklus sollte der Wert der Eingabesteuerung auf ein gesetztes Bit 18 überprüft werden. In diesem Fall war die Datenübertragung nicht erfolgreich und die eingegebenen Daten wurden verworfen. Auch ein solcher fehlerhafter Zyklus muss durch einen Schreibzugriff auf die Eingabesteuerung (nicht Null) beendet werden. Dieser Fehler tritt nur auf, wenn der Adressat innerhalb des RemFPGA die Daten mangels Speicherplatz nicht empfangen konnte. Der Zustand des Adressaten innerhalb des RemFPGA sollte daher zuvor durch die Auslese des Registers IN_STATUS überprüft werden. Dies gilt natürlich auch für die Dateneingabe über den Pipelinebus.

Im Unterschied zum Pipelinebus können beide FIFOs des VME-Busses gleichzeitig aktiv sein.

Kapitel 7

Synthese und Test des RemFPGA

Das Präprozessormodul, welches der RemFPGA steuern wird, existiert zum Zeitpunkt dieser Diplomarbeit noch nicht als funktionierende Hardware. Daher erfolgten sämtliche Tests des Programmcodes ausschließlich mit der Hilfe von Software-Simulationswerkzeugen.
Auch eine Synthese des RemFPGA in seiner endgültigen Form war nicht möglich, da externe Parameter wie die Pinbelegung des FPGA noch nicht bekannt waren. Es ist noch nicht vollständig geklärt, ob auf dem PPM für die seriellen Leitungen zu den PPrASICs differentielle LVDS-Signale oder unipolare Signale verwendet werden sollen, was eine abschließende Synthese der RemFPGA ebenfalls ausschließt.
Jedoch erfordert eine Anpassung an diese Parameter lediglich eine Modifikation des Top-Level Moduls RemTop. Der eigentliche Programmcode des RemFPGA bleibt von diesen Änderungen unberührt.

7.1 Synthese des RemFPGA

Die Synthese eines Xilinx FPGA Designs besteht aus mehreren Schritten. Der erstellte Verilogcode wird zunächst in eine so genannte Netzliste kompiliert. Hierzu wurde der *FPGA Compiler II* von Synopsys [13] verwendet. Die erzeugte Netzliste wurde daraufhin mit Hilfe der Xilinx Software *ISE Alliance 4.1i* ([15] und [16]) in den so genannten *Translate*, *Map* und *Place-and-Route* Schritten in eine weitere, detailliertere Netzliste umgewandelt. Diese neue Netzliste weist jedem internen Baustein des FPGA-Chips eine spezifische Funktion zu und verwirklicht somit das in Verilog definierte Design mit Hilfe der verfügbaren Hardware.

7.1.1 Kompilation mit FPGA Compiler II

Innerhalb der grafischen Bedienoberfläche des FPGA Compiler II muss zunächst ein Projekt angelegt werden und sämtliche Quelldateien des RemFPGA zu diesem hinzugefügt werden. Dies umfasst neben dem eigentlichen Programmcode des Designs auch Quellencode, welcher von der Xilinx Software ISE Alliance 4.1i erzeugt wurde. Die Hardware-spezifischen Komponenten wie SelectRAM oder die Eingabe/Ausgabeblöcke (IOB) des Virtex werden auf diese Weise in den RemFPGA Programmcode eingebunden. Für die Generierung dieses Quellcodes dient das ISE Programm *Core Generator*.
Der Compiler benötigt darüber hinaus noch Informationen wie die verwendete Hardware (Xilinx Virtex-E 1000) oder die maximale Taktrate des Designs. Weiterhin können optionale Randbedingungen für den Compiler definiert werden (so genannte *Constraints*). Einige dieser Constraints beeinflussen den Compiler in seiner Arbeitsweise, andere werden lediglich in die erzeugte Netzliste eingetragen und von späteren Arbeitsschritten wie dem Place-and-Route verwendet.
Constaints umfassen viele verschiedene Angaben wie zum Beispiel Pinzuordnungen, Positionen von Speicherblöcken, besondere Taktfrequenzen für bestimmte Bereiche des Designs und vieles mehr. Diese Angaben können innerhalb des Compilers gemacht werden, genauso können sie jedoch im Top-Level Modul des Designs spezifiziert werden oder in einer zusätzlichen Textdatei oder auch bei der Verwendung anderer Werkzeuge wie zum Beispiel dem Place-and-Route Schritt. Innerhalb der Xilinx Software gibt es ein spezielles visuelles Werkzeug zur Definition beliebiger Constraints, den so genannten *Floor Planer*. Letztendlich steht es dem Anwender frei, wo und wie er diese Randbedingungen spezifiziert.

Die Festlegung der meisten Constraints kann wegen der erwähnten Unkenntnis der wichtigsten Randparameter nicht im Rahmen dieser Diplomarbeit erfolgen und bleibt einem späteren Benutzer überlassen. Grundsätzlich wurde jedoch bei der Erstellung des Programmcodes des RemFPGA strikt darauf geachtet, so wenig proprietäre Softwarewerkzeuge wie möglich zu verwenden. So weit möglich, wurden sämtliche Definitionen und Einstellungen innerhalb des Quellcodes in Textform vorgenommen. Somit wird empfohlen, die zusätzlichen Constraints soweit möglich im Top-Level Modul RemTop des RemFPGA zu platzieren. Als weiteres komfortables Werkzeug bietet sich der Floor Planer an, die Verwendung des Constraints-Editors des FPGA Compiler II sollte dagegen minimiert werden.

7.1.2 Mapping des Designs

Nach der Kompilation durch den FPGA Compiler II wird die erstellte Netzliste im so genannten *Transform*-Schritt in die Xilinx Softwareumgebung importiert. Anschließend erfolgt das *Mapping*, in welchem dem in der Netzliste definierten Design die verfügbaren Ressourcen des Virtex-E 1000 zugewiesen werden. Nach diesem Schritt ist die Auslastung der FPGA-Hardware bekannt.

Aus unbekannten Gründen ergab sich als Resultat der Analyse einige wenige Male eine gravierende Erhöhung der FPGA-Ressourcen. Ein Auszug der *Mapping Logdatei* eines solchen Falles ist unten aufgeführt. Bei komplett neuen *Kompilierungs*- und *Mapping*-Schritten nach geringen, von der Funktionalität her unerheblichen Änderungen des Quellcodes ist dieses Phänomen wieder verschwunden und später nicht reproduzierbar wieder aufgetaucht. Der Autor vertritt die Auffassung, dass dies eventuell von FPGA Compiler II verursacht wird, da dieser nach einem Syntheseschritt manchmal unverständlich viel Arbeitsspeicher belegt und selbst nach Abschluss der Kompilierung nicht wieder freigibt. Auch musste grundsätzlich die ältere von zwei verfügbaren, verschiedenen internen Verilog-Compilern des FPFA Compiler II verwendet werden, da die neuere Version nachweislich Probleme bei späteren *Optimierungs*-Schritt des FPGA Compiler II verursachte. Die ältere Version führt dagegen zum stets erfolgreichen Abschluss der Kompilierungsvorgangs. Da das *Mapping*-Phänomen jedoch erst wenige Tage vor Abschluss dieser Diplomarbeit auftrat, konnte die letztendliche Ursache nicht mehr ermittelt werden.

```
Release 4.1i - Map E.30
Xilinx Mapping Report File for Design 'RemTop'

Design Information
------------------
Command Line   : map -p xcv1000e-bg560-6 -cm area -k 4 -c 100 -tx off remtop.ngd
Target Device  : xv1000e
Target Package : bg560
Target Speed   : -6
Mapper Version : virtexe -- $Revision: 1.58 $
Mapped Date    : Tue Apr 09 08:32:40 2002

Design Summary
--------------
   Number of errors:      1
   Number of warnings:    1
   Number of Slices:            24,746 out of 12,288  201%
   Number of Slices containing
      unrelated logic:           1,502 out of 24,746    6%
   Number of Slice Flip Flops:  16,035 out of 24,576   65%
   Total Number 4 input LUTs:   43,178 out of 24,576  175%
      Number used as LUTs:                    41,886
      Number used as a route-thru:             1,292
   Number of bonded IOBs:          243 out of    404   60%
   Number of Block RAMs:            88 out of     96   91%
   Number of GCLKs:                  3 out of      4   75%
   Number of GCLKIOBs:               3 out of      4   75%
   Number of DLLs:                   3 out of      8   37%
   Number of Startups:               1 out of      1  100%
```

```
Total equivalent gate count for design:  1,893,322
Additional JTAG gate count for IOBs:  11,808

Table of Contents
-----------------
Section 1 - Errors
Section 2 - Warnings
Section 3 - Informational
Section 4 - Removed Logic Summary
Section 5 - Removed Logic
Section 6 - IOB Properties
Section 7 - RPMs
Section 8 - Guide Report
Section 9 - Area Group Summary
Section 10 - Modular Design Summary

Section 1 - Errors
------------------
ERROR:Pack:18 - The design is too large for the given device and package.

   Please check the Design Summary section to see which resource requirement for

   your design exceeds the resources available in the device.



If the slice count exceeds device resources you might try to disable

   register ordering (-r).  Also if your design contains AREA_GROUPs, you may be

   able to improve density by adding COMPRESSION to your AREA_GROUPs if you

   haven't done so already.
```

Bei „normal“ verlaufendem *Mapping* benötigt der RemFPGA knapp 11.000 der verfügbaren Virtex SLICES.

7.1.3 Place-and-Route

Der so genannte *Place-and-Route* Schritt die im *Mapping*-Schritt ermittelten, nötigen FPGA-Ressourcen auf die verfügbare Hardware des Chips. Dabei wird die Plazierung und Verbindung der einzelnen Kompontenten anhand vielerlei (teilweise von Xilinx patentierten) Regeln optimiert, um eine möglichst synchrone Zusammenarbeit aller FPGA-Ressourcen innerhalb des Chips zu erreichen.
Nach diesem Schritt kann ein so genanntes *Bit File* erstellt werden, welches direkt in die Hardware geladen werden kann und den Chip so für seine zugedachte Aufgabe konfiguriert.

7.2 Simulation und Test des RemFPGA

Die Simulation und der Test des RemFPGA erfolgte mit Hilfe der Software *ModelSim 5.5f* von Model Technology [14].
Für die verschiedenen Tests wurde eine modulare Testbench in rein in Verilog entwickelt. Sie simuliert die fünf verschiedenen Schnittstellen des RemFPGA (Pipeline, VME, seriell, I²C und SPI) und erlaubt die Erzeugung beliebiger Zugriffzyklen über VME oder Pipelinebus.

Die seriellen Schnittstellen der PPrASICs werden mit Hilfe von Programmcode simuliert, welcher direkt dem Verilogdesign des PPrASIC entnommen wurde. Somit ist eine perfekte Übereinstimmung mit der reellen Hardware gewährleistet. Für die Simulation von I²C Slave Geräten wurde ein Testmodell verwendet, welches ebenfalls im Quellcode von www.OpenCores.org [11] enthalten ist. Es wurde leicht angepasst, um den Betreib mehrerer I²C-Slaves an einem Bus zu ermöglichen.
ModelSim erlaubt die Simulation mit Hilfe der Daten einer Netzliste wie sie am Ende der FPGA Synthese entstehen. Eine solche timing back-annotated Simulation berücksichtigt sämtliche Signalverzögerungen durch Leitungslängen innerhalb des FPGA und liefert somit eine sehr realistische Simulation der Hardware.
Die unten aufgeführten Simulationsergebnisse wurden jedoch mit einer normalen statischen Timing-Simulation erstellt, die Unterschiede bei den folgenden Beispielen sehr gering sind, die Simulationsgeschwindigkeit jedoch um etwa den Faktor 30 bis 40 absinkt. Auch stellt sich wegen der oben erwähnten seltsamen *Mapping*-Fehlschlägen die Frage, wie zuverlässig die Synthese des Designs ist. Aus diesem Grund wurde auf die Simulation auf RTL-Level zurückgegriffen. Es bleibt jedoch zu sagen, dass sich der RemFPGA nach erfolgreichem *Mapping* und *Place*-andRoute bisher fehlerfrei simulieren ließ.
Die Darstellungen der Simulationen (auch *Wave* genannt) werden im Folgenden jeweils zusammen mit der gleichzeitigen Logging-Ausgabe der Testbench dargestellt.

7.2.1 Dekodierung des Eingangssignals einer seriellen Schnittstelle

Der folgende Ausschnitt zeigt den Empfang eines ReadBACK-Wortes, gefolgt von einem ReadOUT-Header, einem BCID-Wert und zwei RAW-Werten. Das Modul RemSerSyncDecode ist mit der zu verarbeitenden Anzahl von BCID- und RAW-Werten konfiguriert (ganz oben). Direkt darunter sind die Signale der seriellen Leitung zu sehen. Weiter unter sind die Signale aufgeführt, welche ein empfangenes Wort identifizieren und weiterleiten.

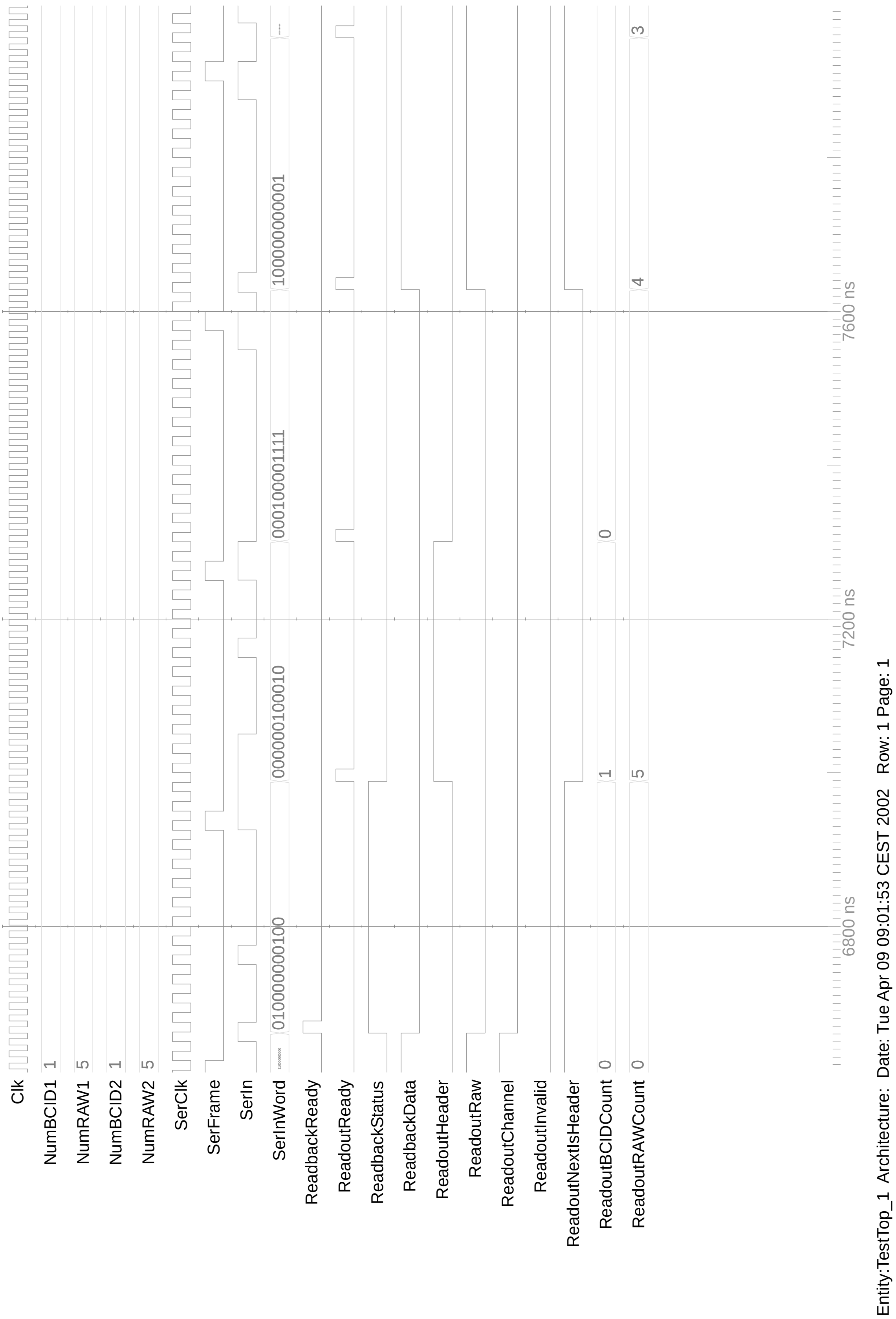


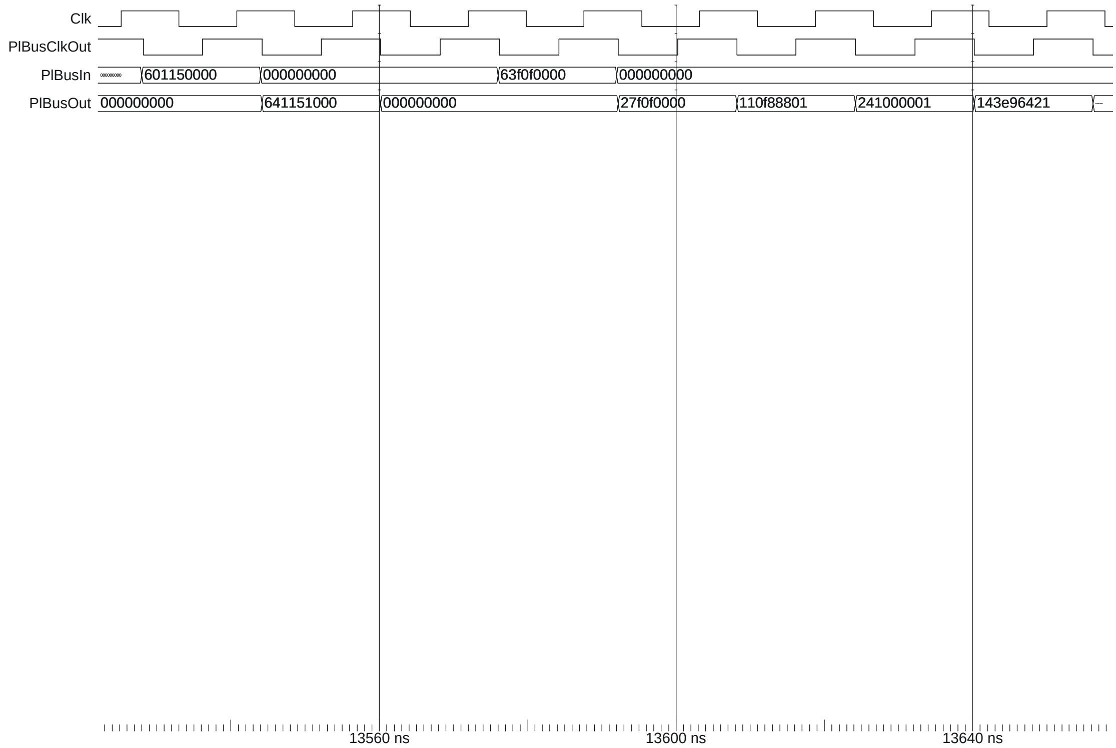
Abbildung 41 - serieller Dateneingang über das Modul RemSerSyncDecode

7.2.2 Auslese von Readout-Daten über den Pipelinebus

Es folgt die Logging-Ausgabe und ein Wave-Schaubild, welches den obersten Teil des loggten Vorgangs darstellt.

```
# Getting ReadOUT data via PLBus
# PlBus: GetDataOutInfo - Addr 01, Arg 0001000000000000, Accept 1
# PlBus: BeginOfData - Addr 3f, Arg 0000, Accept 1
# PlBus: Data 00000000 - bin: 00010000111110001000100000000001, hex: 10f88801
# PlBus: Data 00000001 - bin: 01000011111010010110010000100001, hex: 43e96421
# PlBus: Data 00000002 - bin: 10101000001110111100100010001000, hex: a83bc888
# PlBus: Data 00000003 - bin: 10010110010000100001000100001111, hex: 9642110f
# PlBus: Data 00000004 - bin: 10111100100010001000010000111110, hex: bc88843e
# PlBus: Data 00000005 - bin: 00100001000100001111101010000011, hex: 2110fa83
# PlBus: Data 00000006 - bin: 10001000010000111110100101100100, hex: 8843e964
# PlBus: Data 00000007 - bin: 00001111101010000011101111001000, hex: 0fa83bc8
# PlBus: Data 00000008 - bin: 00111110100101100100001000010001, hex: 3e964211
# PlBus: Data 00000009 - bin: 10000011101111001000100010000100, hex: 83bc8884
# PlBus: Data 0000000a - bin: 01100100001000010001000011111010, hex: 642110fa
# PlBus: Data 0000000b - bin: 11001000100010000100001111101001, hex: c88843e9
# PlBus: Data 0000000c - bin: 00010001000011111010100000111011, hex: 110fa83b
# PlBus: Data 0000000d - bin: 10000100001111101001011001000010, hex: 843e9642
# PlBus: Data 0000000e - bin: 11111010100000111011110010001000, hex: fa83bc88
# PlBus: Data 0000000f - bin: 11101001011001000010000100010000, hex: e9642110
# PlBus: Data 00000010 - bin: 00111011110010001000100001000011, hex: 3bc88843
# PlBus: Data 00000011 - bin: 01000010000100010000111110101000, hex: 42110fa8
# PlBus: Data 00000012 - bin: 10001000100001000011111010010110, hex: 88843e96
# PlBus: Data 00000013 - bin: 00010000111110101000001110111100, hex: 10fa83bc
# PlBus: Data 00000014 - bin: 01000011111010010110010000100001, hex: 43e96421
# PlBus: Data 00000015 - bin: 10101000001110111100100010001000, hex: a83bc888
# PlBus: Data 00000016 - bin: 10010110010000100001000100001111, hex: 9642110f
# PlBus: Data 00000017 - bin: 10111100100010001000010000111110, hex: bc88843e
# PlBus: Data 00000018 - bin: 00100001000100001111101010000011, hex: 2110fa83
# PlBus: Data 00000019 - bin: 10001000010000111110100101100100, hex: 8843e964
# PlBus: Data 0000001a - bin: 00001111101010000011101111001000, hex: 0fa83bc8
# PlBus: Data 0000001b - bin: 00111110100101100100001000010001, hex: 3e964211
# PlBus: Data 0000001c - bin: 10000011101111001000100010000100, hex: 83bc8884
# PlBus: Data 0000001d - bin: 01100100001000010001000011111010, hex: 642110fa
# PlBus: Data 0000001e - bin: 11001000100010000100001111101001, hex: c88843e9
# PlBus: Data 0000001f - bin: 00010001000011111010100000111011, hex: 110fa83b
# PlBus: Data 00000020 - bin: 10000100001111101001011001000010, hex: 843e9642
# PlBus: Data 00000021 - bin: 11111010100000111011110010001000, hex: fa83bc88
# PlBus: Data 00000022 - bin: 11101001011001000010000100010000, hex: e9642110
# PlBus: Data 00000023 - bin: 00111011110010001000100001000011, hex: 3bc88843
# PlBus: Data 00000024 - bin: 01000010000100010000111110101000, hex: 42110fa8
# PlBus: Data 00000025 - bin: 10001000100001000011111010010110, hex: 88843e96
# PlBus: Data 00000026 - bin: 00010000111110101000001110111100, hex: 10fa83bc
# PlBus: Data 00000027 - bin: 01000011111010010110010000100001, hex: 43e96421
# PlBus: Data 00000028 - bin: 10101000001110111100100010001000, hex: a83bc888
# PlBus: Data 00000029 - bin: 10010110010000100001000100001111, hex: 9642110f
# PlBus: Data 0000002a - bin: 10111100100010001000010000111110, hex: bc88843e
# PlBus: Data 0000002b - bin: 00100001000100001111101010000011, hex: 2110fa83
# PlBus: Data 0000002c - bin: 10001000010000111110100101100100, hex: 8843e964
# PlBus: Data 0000002d - bin: 00001111101010000011101111001000, hex: 0fa83bc8
# PlBus: Data 0000002e - bin: 00111110100101100100001000010001, hex: 3e964211
# PlBus: Data 0000002f - bin: 10000011101111001000100010000100, hex: 83bc8884
# PlBus: Data 00000030 - bin: 01100100001000010001000011111010, hex: 642110fa
# PlBus: Data 00000031 - bin: 11001000100010000100001111101001, hex: c88843e9
# PlBus: Data 00000032 - bin: 00010001000011111010100000111011, hex: 110fa83b
# PlBus: Data 00000033 - bin: 10000100001111101001011001000010, hex: 843e9642
# PlBus: Data 00000034 - bin: 11111010100000111011110010001000, hex: fa83bc88
# PlBus: Data 00000035 - bin: 11101001011001000010000100010000, hex: e9642110
# PlBus: Data 00000036 - bin: 00111011110010001000100001000011, hex: 3bc88843
# PlBus: Data 00000037 - bin: 01000010000100010000111110101000, hex: 42110fa8
# PlBus: Data 00000038 - bin: 10001000100001000011111010010110, hex: 88843e96
# PlBus: Data 00000039 - bin: 00010000111110101000001110111100, hex: 10fa83bc
# PlBus: Data 0000003a - bin: 01000011111010010110010000100001, hex: 43e96421
# PlBus: Data 0000003b - bin: 10101000001110111100100010001000, hex: a83bc888
# PlBus: Data 0000003c - bin: 10010110010000100001000100001111, hex: 9642110f
# PlBus: Data 0000003d - bin: 10111100100010001000010000111110, hex: bc88843e
# PlBus: Data 0000003e - bin: 00100001000100001111101010000011, hex: 2110fa83
# PlBus: Data 0000003f - bin: 10001000010000111110100101100100, hex: 8843e964
```

```
# PlBus: Data 00000040 - bin: 00001111101010000011101111001000, hex: 0fa83bc8
# PlBus: Data 00000041 - bin: 00111110100101100100001000010001, hex: 3e964211
# PlBus: Data 00000042 - bin: 10000011101111001000100010000100, hex: 83bc8884
# PlBus: Data 00000043 - bin: 01100100001000010001000011111010, hex: 642110fa
# PlBus: Data 00000044 - bin: 11001000100010000100001111101001, hex: c88843e9
# PlBus: Data 00000045 - bin: 00010001000011111010100000111011, hex: 110fa83b
# PlBus: Data 00000046 - bin: 10000100001111101001011001000010, hex: 843e9642
# PlBus: Data 00000047 - bin: 11111010100000111011110010001000, hex: fa83bc88
# PlBus: Data 00000048 - bin: 11101001011001000010000100010000, hex: e9642110
# PlBus: Data 00000049 - bin: 00111011110010001000100001000011, hex: 3bc88843
# PlBus: Data 0000004a - bin: 01000010000100010000111110101000, hex: 42110fa8
# PlBus: Data 0000004b - bin: 10001000100001000011111010010110, hex: 88843e96
# PlBus: Data 0000004c - bin: 00000000000000101000001110111100, hex: 000283bc
# PlBus: EndOfData - Addr 01, Arg 0001, Accept 1
```



Entity:TestTop_1 Architecture: Date: Tue Apr 09 09:13:22 CEST 2002 Row: 1 Page: 1

Abbildung 42 - Datenverkehr über den Pipelinebus

7.2.3 Auslese von ReadBACK-Daten über VME

Es folgt das Log einer Auslese von ReadBACK-Daten über VME. Der Auslesezyklus wurde dabei ausgelöst durch das konfigurierbare VME Interrupt-Signal des RemFPGA. In diesem Fall ist das Signal auf die Meldung von verfügbaren ReadBACK-Daten konfiguriert.

```
# Getting ReadBACK data via VME
#   VME R_REG_DATA_STATUS Status: 11001111111111101000000000010001, cffe8011
#   VME R_REG_RB_DATAREADY Status: 00000000000000000001000111111111, 000011ff
#   VME R_REG_RB_MEMFULL Status: 00000000000000000000000000000000, 00000000
# VME: FIFO status - bin: 00000000000001100000000000000000, hex: 00060000
# VME: ReadBACK started, length of data: 0011
# VME: ReadBACK 00000000 - bin: 00000001000100000000000001000010, hex: 01100042
# VME: ReadBACK 00000001 - bin: 00000000000000110000000010000010, hex: 00030082
# VME: ReadBACK 00000002 - bin: 00000000000000000000101111111111, hex: 00000bff
# VME: ReadBACK 00000003 - bin: 00000000000000110000100010000010, hex: 00030882
# VME: ReadBACK 00000004 - bin: 00000000000000000000101111111111, hex: 00000bff
# VME: ReadBACK 00000005 - bin: 00000000000000110001000010000010, hex: 00031082
# VME: ReadBACK 00000006 - bin: 00000000000000000000101111111111, hex: 00000bff
# VME: ReadBACK 00000007 - bin: 00000000000000110001100010000010, hex: 00031882
# VME: ReadBACK 00000008 - bin: 00000000000000000000101111111111, hex: 00000bff
# VME: ReadBACK 00000009 - bin: 00000000000000110010000010000010, hex: 00032082
# VME: ReadBACK 0000000a - bin: 00000000000000000000101111111111, hex: 00000bff
# VME: ReadBACK 0000000b - bin: 00000000000000110010100010000010, hex: 00032882
# VME: ReadBACK 0000000c - bin: 00000000000000000000101111111111, hex: 00000bff
# VME: ReadBACK 0000000d - bin: 00000000000000110011000010000010, hex: 00033082
# VME: ReadBACK 0000000e - bin: 00000000000000000000101111111111, hex: 00000bff
# VME: ReadBACK 0000000f - bin: 00000000000000110011100010000010, hex: 00033882
# VME: ReadBACK 00000010 - bin: 00000000000000000000101111111111, hex: 00000bff
# VME: ReadBACK status - bin: 00000000000001000000000000010001, hex: 00040011
# VME: ReadBACK cycle successfully completed
#   VME R_REG_DATA_STATUS Status: 11001111111111100000000000000000, cffe0000
#   VME R_REG_RB_DATAREADY Status: 00000000000000000000000000000000, 00000000
#   VME R_REG_RB_MEMFULL Status: 00000000000000000000000000000000, 00000000
#   VME R_REG_RB_DATALOSS Status: 00000000000000000000000000000000, 00000000
```

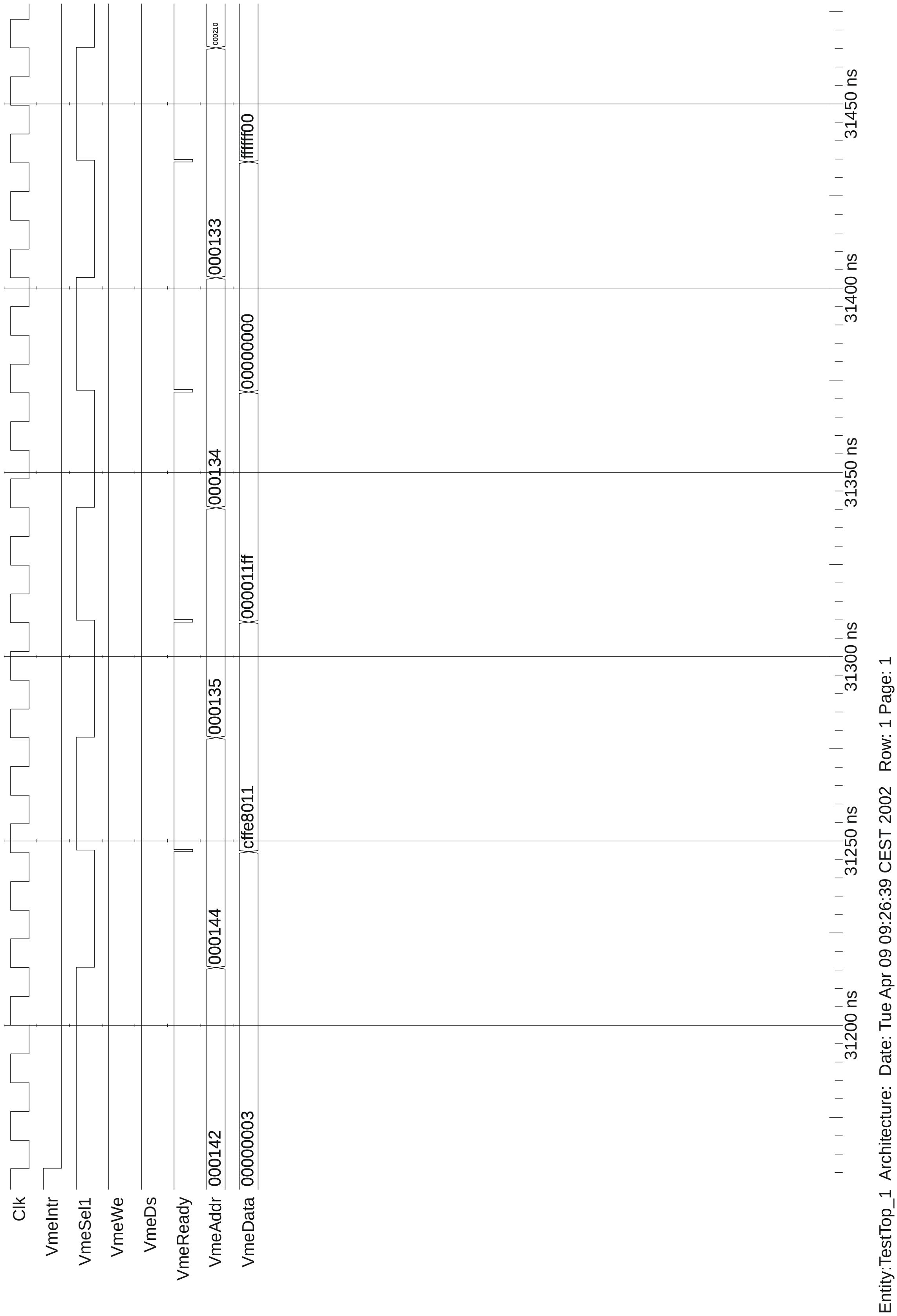


Abbildung 43 - ReadBACK-Auslese über VME - Teil 1

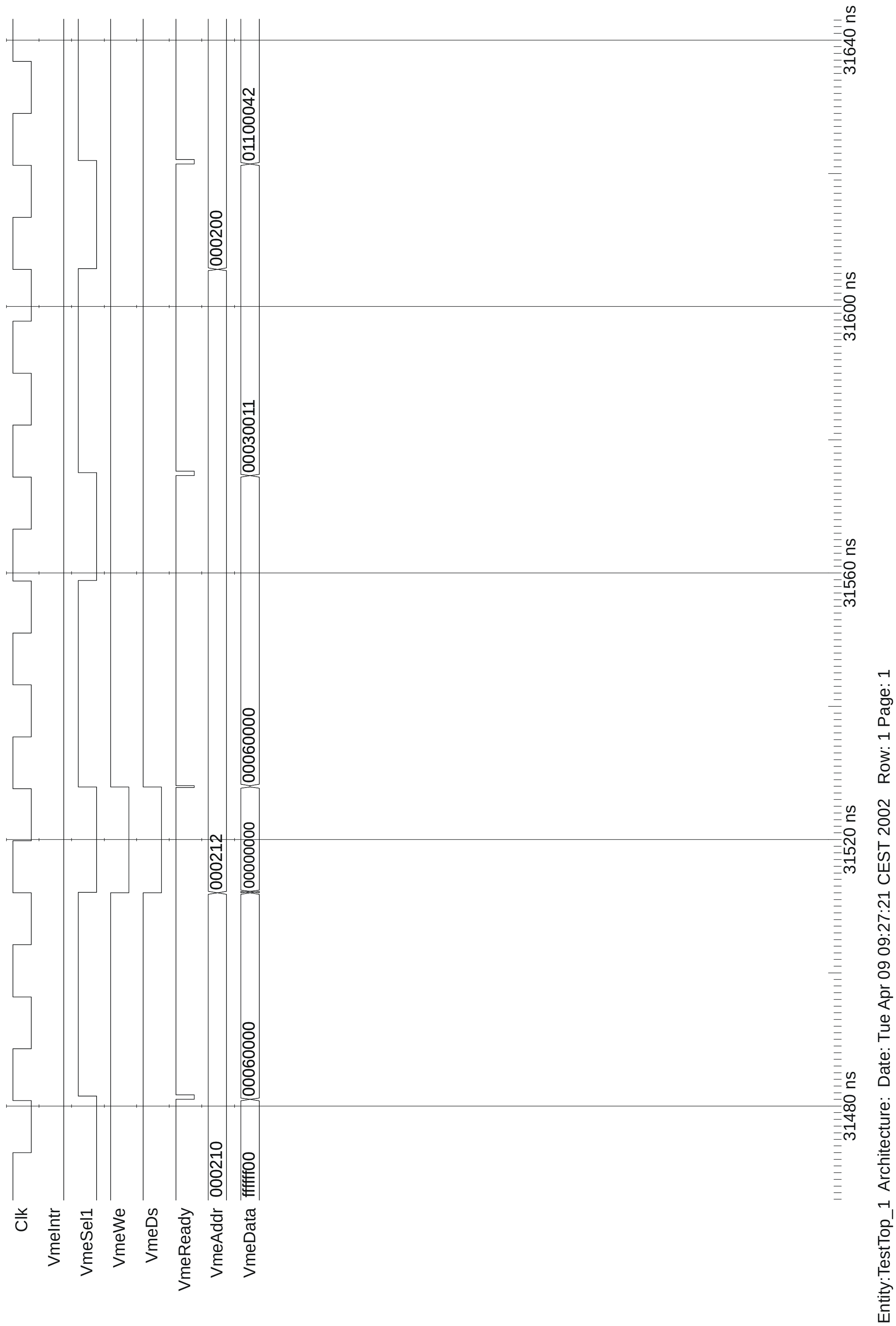


Abbildung 44 - ReadBACK-Auslese über VME - Teil 2

7.2.4 Verwendung des I²C-Busses

Zuletzt wird der Einsatz des I²C-Masters von OpenCores.org demonstriert. An der RemFPGA angeschlossen sind zwei Slave-Geräte mit den Adressen (’b0010000 und ’b0010001). Die gestrichelten Linien in dem *Wave*-Schaubild stellen den Effekt eines innerhalb des FPGA instanzierten PullUp-Elements dar. Auf diese Weise muss keine externere Schaltung für diese Funktion des I²C-Busses verwendet werden.

```
# Writing 5 words onto I2C via VME
#   VME FIFO Status: 00000000000001110000000000000000, 00070000
#   VME Input Status: 00000000000000000000000000000000, 00000000
#   VME starting Input cycle
#   VME Input Status: 00000000000000110000000000000000, 00030000
#   VME Input Status: 00000000000000100000000000000000, 00020000
#   VME Input: writing data (six 16 Bit words -> three 32 Bit VME words
# VME: Wrote to 000201 - bin: 00010001011001100000000001100101,  hex: 11660065
# VME: Wrote to 000201 - bin: 00010001111111110001000011111111,  hex: 11ff10ff
# VME: Wrote to 000201 - bin: 00010000111100000001000101010101,  hex: 10f01155
#   VME Input Status: 00000000000000100000000000000011, 00020003
#   VME completing Input cycle
#   VME Input Status: 00000000000000000000000000000011, 00000003
#   VME Input Status: 00000000000000000000000000000011, 00000003
# DEBUG i2c_slave 10; start condition detected at              2747000
# DEBUG i2c_slave 11; start condition detected at              2747000
# DEBUG i2c_slave 11; command byte received (write) at              4028000
# DEBUG i2c_slave 11; data write 66
# DEBUG i2c_slave 10; stop condition detected at              5836000
# DEBUG i2c_slave 11; stop condition detected at              5836000
# DEBUG i2c_slave 10; start condition detected at              5992000
# DEBUG i2c_slave 11; start condition detected at              5992000
# DEBUG i2c_slave 10; command byte received (write) at              7273000
# DEBUG i2c_slave 10; data write ff
# DEBUG i2c_slave 10; stop condition detected at              9080000
# DEBUG i2c_slave 11; stop condition detected at              9080000
# DEBUG i2c_slave 10; start condition detected at              9236000
# DEBUG i2c_slave 11; start condition detected at              9236000
# DEBUG i2c_slave 11; command byte received (write) at             10518000
# DEBUG i2c_slave 11; data write ff
# DEBUG i2c_slave 10; stop condition detected at             12325000
# DEBUG i2c_slave 11; stop condition detected at             12325000
# DEBUG i2c_slave 10; start condition detected at             12481000
# DEBUG i2c_slave 11; start condition detected at             12481000
# DEBUG i2c_slave 11; command byte received (write) at             13762000
```

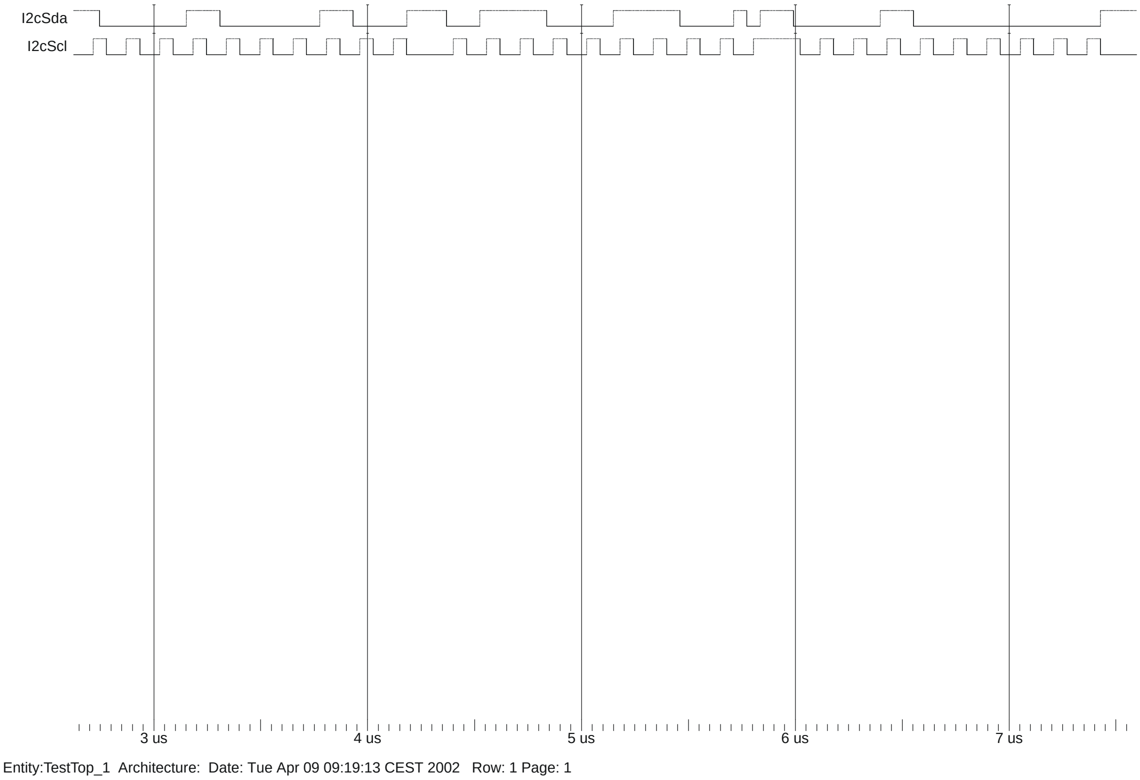


Abbildung 45 - I²C Kommunikation

Kapitel 8

Zusammenfassung

Diese Arbeit hat die Entwicklung der Funktionalität des Readout-Merger zum Mittelpunkt. Dieser Baustein dient erstens der zentralen Steuerung aller Komponenten des Präprozessormoduls und zweitens dem Empfang und der Weiterleitung von ReadOUT-Daten der PPrASICs.
Um veränderten Anforderungen der Zukunft Rechnung tragen zu können, wird für diese Funktionalität ein FPGA-Chip verwendet. Folglich muss der im Rahmen dieser Diplomarbeit erstelle Programmcode möglichst modular aufgebaut sein, um zukünftige Änderungen mit Hilfe von minimalem Arbeitsaufwand zu ermöglichen.
Es wurde lediglich die Programmierung des RemFPGA entwickelt und ausführlich getestet. Die umgebende Hardware ist zum Zeitpunkt dieser Arbeit noch nicht existent, weshalb der RemFPGA nicht in der Praxis getestet werden konnte.
Sämtliche Tests waren mit Ausnahme der seltsamen *Mapping*-Probleme (siehe Abschnitt 7.1.2) fehlerlos und haben keine Probleme des erstellten Programmcodes aufzeigen können. Die wenige Tage vor Abschluss der Arbeit zum ersten Mal aufgetretenen *Mapping*-Fehler müssen jedoch weiter untersucht werden. Die logische Korrektheit des erstellten Programmcodes wurde im Rahmen dieser Diplomarbeit ausführlich getestet und analysiert. Es muss jedoch noch abschließend verifiziert werden, ob das erstellte Design auch mit der verfügbaren Kapazität des Virtex-E 1000 realisiert werden kann.

8.1 Abschätzung der ReadOUT-Datenrate

Die wichtigste Funktion des RemFPGA neben der Konfiguration des Präprozessormoduls ist die Kompression von ReadOUT-Daten und die Übertragung über den Pipelinebus. Die Betrachtungen in älteren Publikationen zu Kompressionsleistungen und zu erwartenden Datenraten sind aufgrund von Veränderungen im Design des PPrASIC überholt. Die Übertragung von drei Flagbits mit jedem BCID-Wert und von einem Flagbit mit jedem RAW-Wert konnte in den älteren Untersuchungen noch nicht berücksichtigt werden. Auch wurde die Existenz des BCID-Wertes von den meisten Arbeiten nicht betrachtet oder komplett ignoriert.
Im Folgenden werden einige detaillierte Kalkulationen zu den zu erwartenden Datenraten vorgestellt. Dabei werden zwei unterschiedliche Level-1 Accept Raten (100kHz und 75kHz) und verschiedene Mengen an ausgelesenen BCID- und RAW-Werten berücksichtigt. In diesen Berechnungen werden sehr konservative Werte für die Güte der Kompression verwendet. RAW-Werte werden beispielsweise mit einer Rate von nur 2,2 komprimiert, während die zu erwartende Rate bei mindestens 2,4 liegen dürfte (siehe Kapitel 6.1).
In der zweitletzten Spalte ist die Datenrate bei deaktivierter Kompression aufgeführt, die letzte Spalte enthält die komprimierten Zahlenwerte. Es wird von einer fehlerfreien Funktion der PPrASICs ausgegangen, daher ist der variable Kanal-Header nur ein Bit breit. Der Header des gesamten Datenstroms verfügt über 18 Flagbits. Jede Slave-Node schließt die Ausgabe mit einem *EndOfData*-Befehl ab, daher 32 Bits pro Präprozessormodul. Der Master muss *leere Wörter* ausgeben, bis die *EndOfData*-Befehle aller acht PPMs angekommen sind, dies bedingt 9*32=288 Bits. Hinzu kommen der *BeginOfData*-Befehl am Anfang der Sequenz und zwei weitere Taktsignale, die der ROD zur Verarbeitung der Daten benötigt.

Data type	Bits	Multiplier	Total Bits	Compress	MB/s	MB/s
BCID Sample	8	512	4096	2	36.62	18.31
BCID Algorithm	3	512	1536	1	13.73	13.73
RAW Sample	0	512	0	2.2	0	0
RAW Ext BCID	0	512	0	1	0	0
Flags (per channel)	1	512	512	1	4.58	4.58
Flags (per PPM)	18	8	144	1	1.29	1.29
PBus (per PPM)	32	8	256	1	2.29	2.29
PBus (per ROD)	384	1	384	1	3.43	3.43
Data rate: 75 kHz			**6928**	**1.42**	**61.94**	**43.63**

Abbildung 46 - 1 BCID, 0 RAW, Level-1 Rate 75kHz

Data type	Bits	Multiplier	Total Bits	Compress	MB/s	MB/s
BCID Sample	8	512	4096	2	36.62	18.31
BCID Algorithm	3	512	1536	1	13.73	13.73
RAW Sample	30	512	15360	2.2	137.33	62.42
RAW Ext BCID	5	512	2560	1	22.89	22.89
Flags (per channel)	1	512	512	1	4.58	4.58
Flags (per PPM)	18	8	144	1	1.29	1.29
PBus (per PPM)	32	8	256	1	2.29	2.29
PBus (per ROD)	384	1	384	1	3.43	3.43
Data rate: 75 kHz			**24848**	**1.72**	**222.16**	**128.94**

Abbildung 47 - 1 BCID, 3 RAW, Level 1 Rate 75kHz

Data type	Bits	Multiplier	Total Bits	Compress	MB/s	MB/s
BCID Sample	8	512	4096	2	36.62	18.31
BCID Algorithm	3	512	1536	1	13.73	13.73
RAW Sample	50	512	25600	2.2	228.88	104.04
RAW Ext BCID	5	512	2560	1	22.89	22.89
Flags (per channel)	1	512	512	1	4.58	4.58
Flags (per PPM)	18	8	144	1	1.29	1.29
PBus (per PPM)	32	8	256	1	2.29	2.29
PBus (per ROD)	384	1	384	1	3.43	3.43
Data rate: 75 kHz			**35088**	**1.84**	**313.71**	**170.56**

Abbildung 48 - 1 BCID, 5 RAW, Level 1 Rate 75kHz

Data type	Bits	Multiplier	Total Bits	Compress	MB/s	MB/s
BCID Sample	8	512	4096	2	48.83	24.41
BCID Algorithm	3	512	1536	1	18.31	18.31
RAW Sample	0	512	0	2.2	0	0
RAW Ext BCID	0	512	0	1	0	0
Flags (per channel)	1	512	512	1	6.1	6.1
Flags (per PPM)	18	8	144	1	1.72	1.72
PBus (per PPM)	32	8	256	1	3.05	3.05
PBus (per ROD)	384	1	384	1	4.58	4.58
Data rate: 100 kHz			**6928**	**1.42**	**82.59**	**58.17**

Abbildung 49 - 1 BCID, 0 RAW, Level 1 Rate 100kHz

Data type	Bits	Multiplier	Total Bits	Compress	MB/s	MB/s
BCID Sample	8	512	4096	2	48.83	24.41
BCID Algorithm	3	512	1536	1	18.31	18.31
RAW Sample	30	512	15360	2.2	183.11	83.23
RAW Ext BCID	5	512	2560	1	30.52	30.52
Flags (per channel)	1	512	512	1	6.1	6.1
Flags (per PPM)	18	8	144	1	1.72	1.72
PBus (per PPM)	32	8	256	1	3.05	3.05
PBus (per ROD)	384	1	384	1	4.58	4.58
Data rate: 100 kHz			**24848**	**1.72**	**296.21**	**171.92**

Abbildung 50 - 1 BCID, 3 RAW, Level 1 Rate 100kHz

Data type	Bits	Multiplier	Total Bits	Compress	MB/s	MB/s
BCID Sample	8	512	4096	2	48.83	24.41
BCID Algorithm	3	512	1536	1	18.31	18.31
RAW Sample	50	512	25600	2.2	305.18	138.72
RAW Ext BCID	5	512	2560	1	30.52	30.52
Flags (per channel)	1	512	512	1	6.1	6.1
Flags (per PPM)	18	8	144	1	1.72	1.72
PBus (per PPM)	32	8	256	1	3.05	3.05
PBus (per ROD)	384	1	384	1	4.58	4.58
Data rate: 100 kHz			**35088**	**1.84**	**418.28**	**227.41**

Abbildung 51 - 1 BCID, 5 RAW, Level 1 Rate 100kHz

Das Resultat dieser Berechnungen ist die Erkenntnis, dass der Pipelinebus nicht wie ursprünglich geplant mit einer Frequenz von 40MHz betrieben werden kann (Datenrate 160 MB/s). Für den Transport der anfallenden Datenmenge ist in fast allen Konfigurationen eine Datenrate von 60MHz mit einer Datenrate von 240 MB/s erforderlich. Bei einer maximalen Datenrate von 227,41 MB/s wird eine noch höhere Taktrate nicht benötigt.
Der Pipelinebus wurde in der Vergangenheit mit einer Frequenz von 40MHz erfolgreich getestet [8]. Dieser Test erfolgte mit völlig ungeschützten Leitungen und mit einem ROD-Prototyp und einer Testversion der Readout-Mergers in ASIC-Design (RemASIC). Beide verwendeten ein sehr starres und wenig robustes Timing-Modell für die Taktung des Busses und für die Übertragung von Daten. Im Besonderen wertete der ROD-Prototyp das Eingangssignal synchron mit dem Ausgabetakt des Pipelinebusses aus. Eine relative einfache Anpassung ermöglicht die Taktung der Auswertung des Eingangs mit Hilfe des über die Back-plane übertragenen Taktsignals. Dieses unterliegt derselben Verzögerung durch Leitungseffekte wie das Datensignal. Somit kann sehr einfach ein Synchronisations- und Geschwindigkeitsproblem eliminiert werden.
Weiterhin werden die Datensignale, wie in Abbildung 21 (Seite 33) gezeigt, durch ein Latch-Register geleitet. Die Daten fließen also von der Back-Plane zum RemFPGA und von dort zurück zur Backplane weiter zum nächsten RemFPGA (oder ROD). Das Design des RemFPGA sieht für das Taktsignal denselben Pfad vor (ohne Latch-Register, welches jedoch keine Verzögerung bedingt), um die Signale synchron zu halten. Die Testversionen des RemASIC wurden im Test jedoch mit einem parallelen, unverzögerten Taktsignal gespeist. Zusätzlich arbeitet der Eingang des neuen RemFPGA mit der positiven Flanke des Taktsignals, während neue Daten mit der negativen Flanke ausgegeben werden. Dies wurde vom RemASIC nur unsauber realisiert, was zu einer weiteren Synchronisationsschwierigkeit führte.
Trotz all dieser potentiellen Probleme funktionierte der Pipelinebus im Test mit 40 MHz. Zieht man die Verwendung einer teilweise abgeschirmten Back-Plane und die Elimination zweier erheblicher Synchronisationsprobleme in Betracht, so sollte sich der Pipelinebus nun ohne weitere Probleme mit einer Geschwindigkeit von 60 MHz betreiben lassen.
Ein weiteres potentielles Problem bei der Datenauslese ist die Tatsache, dass die maximale Level-1 Accept Rate nur ein Durchschnittswert ist. Daher können einige Ereignisse auch mit einer deutlich höheren Frequenz auftreten. Der RemFPGA kann intern ReadOUT-Daten von bis zu ungefähr 40 Level-1 Accept Ereignissen speichern. Damit ist auch in diesen Fällen die korrekte Verarbeitung aller Daten sichergestellt.
Somit können alle anfallenden ReadOUT-Daten mit ausreichender Geschwindigkeit verarbeitet und über den Pipelinebus übertragen werden. Dies ist eine wesentliche Anforderung an das Leistungsprofil des RemFPGA.

8.2 Fazit

Während der Auslese von ReadOUT-Daten können parallel dazu über den VME-Bus beliebige Konfigurationsoperationen ausgeführt werden. Auch können die Register und ReadBACK-Daten der PPrASICs ohne Unterbrechung des ReadOUT-Zyklus beliebig ausgelesen und verändert werden.

Weiterhin bieten der VME-Bus und der Pipelinebus die exakt gleiche Funktionalität, alle Bereiche des RemFPGA können beliebig über beide Schnittstellen angesprochen werden. Somit ist auch der Testbetrieb eines RemFPGA oder eines ganzen PPM mit nur einem Kommunikationsbus möglich.

Bei der Entwicklung des RemFPGA wurde auf eine saubere Hierarchie des erstellten Programmcodes Wert gelegt. Dies sichert zusammen mit den im Quellcode enthaltenen Kommentaren die zukünftige Anpassung des RemFPGA an neue Bedürfnisse mit Hilfe eines vertretbaren Arbeitsaufwands. Auch wurde der Quellcode als ausschließlicher Informationsträger verwendet. Die Verwendung von proprietären Werkzeugen mit eigenen, zusätzlichen Projektdateien wurde soweit möglich vermieden.

Die durchgeführten Tests und Simulationen des RemFPGA belegen das ordnungsgemäße Funktionieren des Designs ausführlich. Jedoch bleibt ungeklärt, ob die Kapazität des Virtex-E 1000 eventuell nicht ausreichend für dieses Design ist oder ob es sich um einen Systemfehler oder Anwendungsfehler bei der verwendeten Software handelt.

8.3 Ausblick

Zusammen mit dem Abschluss dieser Arbeit tritt die Entwicklung der anderen Komponenten des Präprozessormoduls ebenfalls in die Endphase ein. Eine Gammaversion des PPrASICs wurde gefertigt und geliefert und auch das zugehörige PPrMCM wurde fertig gestellt und erste Exemplare wurden produziert. Auch die anderen Komponenten des PPM stehen in Vorversionen zur Verfügung. Die einzelnen Komponenten werden nun getestet und ein erster Prototyp des Präprozessormoduls wird erstellt. Diese Arbeit wird noch einige Monate in Anspruch nehmen.

Einer der nächsten Schritte ist das Design des Readout-Driver Moduls ROD. In diesem Bereich muss vor allem der Programmcode des auf dem ROD enthaltenen FPGAs erstellt und getestet werden (ebenfalls ein Xilinx Virtex-E 1000).

Das größere Ziel ist die Durchführung des so genannten SLICE-Tests in Heidelberg. Bei diesem Test werden einige der 7296 Kanäle des gesamten Level-1 Kalorimeter Triggers, angefangen beim analogen Dateneingang bis hin zur Erzeugung des finalen Level-1 Accept Signals, aufgebaut und getestet. Dies dient der Überprüfung des ordnungsgemäßen Zusammenarbeitens aller Komponenten des Kalorimeter Triggers, da die einzelnen Teile in verschiedenen Physikinstituten in ganz Europa entwickelt werden.

Schließlich soll im Jahr 2006 der Large Hadron Collider seine Arbeit in Genf aufnehmen.

Danksagung

Ich möchte mich bei Allen bedanken, die mir durch sachliche Hilfe oder moralische Unterstützung bei der Abfassung meiner Diplomarbeit geholfen haben.

- **Prof. Meier für die Möglichkeit, am KIP meine Diplomarbeit zu erstellen. Weiterhin für die regelmäßige Teilnahme an den Gruppenbesprechungen trotz vieler anderweitiger Verpflichtungen.**
- **Prof. Lindenstruth erklärte sich bereit, die Zweitkorrektur zu übernehmen.**
- **Klaus Schmitt für eine stets freundliche und immer kompetente Hilfestellung bei Problemen aller Art.**
- **Paul Hanke für die zielgerichtete aber trotzdem stets humorvolle Leitung der Gruppenbesprechnungen**
- **Thomas Nirmaier für die stets gute Laune im Institutsraum.**
- **Ralf Achenbach und Kambiz Mahboubi für die Hilfe beim Verständnis des PPr-ASIC und den Humor bei Gruppenbesprechungen.**
- **Robert Weiss für die kompetente und schnelle Hilfe bei Netzwerkproblemen.**
- **Den Vision-Leuten „hinten rechtes“ für die Hilfe bei ModelSim Problemen.**
- **Meinen WG-Mitbewohnern für die Geduld und Rücksichtnahme.**
- **Dem HP LaserJet 8100 DN Drucker für seine schnelle und unermüdliche Arbeit.**

Literaturverzeichnis

[1] **The LFAN-Test Specification –**
http://www.kip.uni-heidelberg.de/atlas/DATA/docs/LFAN_Test.pdf

[2] **Xilinx Corporation** – http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=ss_vir

[3] **Philips I²C Bus –** http://www.semiconductors.philips.com/i2c/

[4] **I²C Spezifikation –**
http://www.semiconductors.philips.com/acrobat/various/I2C_BUS_SPECIFICATION_3.pdf

[5] **Phos4 Spezifikation –** http://bonner-ntserver.rice.edu/cms/phos4.pdf

[6] **Maxim MAX529 –** http://pdfserv.maxim-ic.com/arpdf/MAX528-MAX529.pdf

[7] **PPrASIC –** http://www.kip.uni-heidelberg.de/atlas/docs/chips.html

[8] **B. Stelzer – A Read-Out Driver Prototype**
http://www.kip.uni-heidelberg.de/Veroeffentlichungen/ps/bstelzer.diplom.ps.gz

[9] **B. Niemann – Datenkompression für die Auslese des ATLAS Level-1 Triggers**
http://www.kip.uni-heidelberg.de/Veroeffentlichungen/ps/niemann.diplom.ps.gz

[10] **V. Schatz – Test of a Readout and Compression ASIC**
http://www.kip.uni-heidelberg.de/Veroeffentlichungen/ps/schatz.diplom.ps.gz

[11] **OpenCore I²C-Controller –** http://www.opencores.org/projects/i2c/

[12] **Dan Husmann – Der ATLAS Level-1 Trigger Preprozessor ASIC**
http://www.kip.uni-heidelberg.de/Veroeffentlichungen/ps/husmann.dipl.ps.gz

[13] **Synopsys FPGA Compiler II –** http://www.synopsys.com/products/fpga/fpga_solution.html

[15] **Xilinx ISE 4.1i Dokumentation –** http://support.xilinx.com/support/sw_manuals/xilinx4/

[16] **Xilinx ISE Alliance 4.1i –**
http://support.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Design+Tools

[14] **ModelSim 5.5 SE –** http://www.model.com/products/default.asp

[17] **Povh, Rith, Scholz und Zetsche – Teilchen und Kerne**, Springer Verlag, 3. Aufl.

[18] **ATLAS Level-1 Trigger Homepage –**
http://atlas.web.cern.ch/Atlas/GROUPS/DAQTRIG/LEVEL1/level1.html

[19] **ATLAS Homepage –** http://atlasinfo.cern.ch/Atlas/Welcome.html

[20] **CMS Outreach –** http://cmsinfo.cern.ch/Welcome.html/CMSdocuments/CMSdocuments.html

[21] **ATLAS Homepage des Kirchhoff-Instituts –** http://web.kip.uni-heidelberg.de/atlas/

[22] **PPrASIC User Manual –** http://web.kip.uni-heidelberg.de/atlas/DATA/docs/pprasic/pprasicman.ps